다중스레드 구조에서 함수 언어 루프의 효과적 실행

하 상 호†

요 약

다중스레딩은 계산과 통신을 효과적으로 중첩시킴으로써 메모리 참조 지체시간과 동기화에 따른 문제점을 해결할 수 있다. 함수 언어로 작성된 프로그램으로부터 다중스레드 코드를 생성하기 위한 다양한 컴파일러 기법이 개발되어 왔지만, 루프의 효과적 구현에 관한 연구는 아직 미흡하다. 보통, 다중스레딩으로 루프를 실행시키는 데에는 부담이 따르며, 이러한 부담은 다중스레딩의 효과를 심각하게 떨어뜨릴 수 있다. 본 논문은 기계 구조와 컴파일러의 관점에서 다중스레딩 방식에 의한 루프의 실행을 최적화시킬 수 있는 여러 가지 방법을 제안하고, 행렬 곱셈 프로그램에 대해서 그 방법을 모의실험하고 분석한다.

The Efficient Execution of Functional Language Loops on the Multithreaded Architectures

Sang-Ho Ha†

ABSTRACT

Multithreading is attractive in that it can tolerate memory latency and synchronization by effectively overlapping communication with computation. While several compiler techniques have been developed to produce multithreaded codes from functional languages programs, there still remains a lot of works to implement loops effectively. Executing loops in a style of multithreading usually causes some overheads, which can reduce severely the effect of multirheading. This paper suggests several methods in terms of architectures or compilers which can optimize loop execution by multithreading. We then simulate and analyze them for the matrix multiplication program.

1. 서 론

다중스레딩(multithreading)은 병렬 컴퓨터의 확장성을 근본적으로 제한하고 있는 메모리 참조의 지체시간과 동기화의 문제[1]를 해결할 수 있다는 점에서 최근에 많은 관심을 얻고 있다. 다중스레딩은 기본적으로 통신과 계산을 중첩시킴으로써 실행의 효과를 높이는데 그 목적이 있다. 다중스레딩에서는 원격 메모리에 값을 요청한 후에 그 반응을 대기하지 않고 즉각 다른 스레드(thread)를 계속 실행하고, 이 후에 그 반응이 도달하면 해당 스레드의 실행을 속개하는 방식으로 프로그램을 실행한다. 스레드는 순차적으로 실행되는 명령어의 집합으로 정의되는데, 일반적으로 계산 단위는 프로세스(process)에 비해서 작다. 따라서 스레드간의 문맥교환(context switching)이 빈번히 발생하게 되는데, 다중스레딩이 효과적이기 위해서는 문맥교환에 따른 부담이 최소화되어야 한다. 최근에 메모리 연산의 분

※ 이 연구는 1998학년도 순천향대학교 지원에 의한 결과임.
† 정 회 원 : 순천향대학교 정보기술공학부 교수
논문접수 : 1999년 9월 18일, 심사완료 : 2000년 1월 10일

할 트랜잭션을 허용하고, 신속한 문맥교환을 지원하는 다중스레드 구조 TAM[2], DAVRID[3], StarT-Voyager [4] 등이 연구, 개발되어 오고 있다.

다중스레딩이 효과적이기 위한 다른 조건으로 프로그램이 많은 병렬성을 내포하고 있어야 한다. 최근에 작은 계산단위의 대규모 병렬성을 표현할 수 있는 비정형성 함수 언어 Id[5] 프로그램에 대해서 효율적인 스레드 코드를 생성하는 여러 컴파일 기법[6, 7, 8]이 연구되어 왔으며 특히, Id의 루프 구문을 다중스레딩 환경에서 효과적으로 실행시킬 수 있는 기법[9, 10]이 연구되어 왔다. [10]에 제시된 루프 펼침 기법 KU-Loop은 루프를 여러 개의 프레임에 펼쳐서 병렬로 실행시킨다. 이를 위해서 각 프레임에 여러 개의 루프 상수를 미리 전달하여 루프 실행 환경을 먼저 설정한 후에 루프의 실행을 시작시킨다. 루프 실행 중에 각 프레임에서 현재의 반복문 실행을 종료한 후에 다음번째 반복문을 실행하기 전에 그 프레임을 다시 초기화하는 것이 필요하다. 여기서 루프 실행 환경을 설정하는 작업과 다음번째 반복문의 실행 시작 전에 프레임을 다시 초기화하는 작업은 다중스레딩 방식에 의한 루프 실행을 위해서 새롭게 추가된 부담이다. 이러한 부담은 다중스레딩에 의한 루프의 실행을 위해서 필연적이며, 이러한 실행 부담은 예상한 바와는 다르게 다중스레딩의 효과를 현격하게 감소시킬 수 있다.

본 논문에서는 다중스레드 구조 DAVRID에서 Id 언어 구현시 기계 구조와 컴파일러의 관점에서 다중스레딩에 따른 루프의 실행 부담을 최소화하기 위한 3가지 방법을 제시한다. 첫째, 프레임 초기화에 따른 부담을 시스템의 부하균형과 병렬성을 향상시키는 관점에서 분산시킨다. 둘째, 루프 실행 환경 설정에 따른 부담을 줄인다. 현재는 루프 프레임 설정시에 각 루프 상수가 자신의 메시지를 통해서 전달되는데, 여기서는 가능한 여러 개의 루프 상수를 한 개의 메시지로 통합하여 전달함으로써 메시지 생성, 전달, 처리에 따른 부담을 줄인다. 셋째, 배열 원소에 대한 실제 주소를 계산하는데 있어서 공통 계산 부분을 공유하여 전역 메모리에 대한 참조 횟수를 줄인다. 일반적으로 배열 연산은 루프를 통해서 표현되므로, 배열 연산의 최적화를 통해서 루프 실행을 향상시킬 수 있다. Id 언어에서 각 배열 원소에 대한 주소를 계산할 때 전역 메모리에의 참조를 필요로 한다.

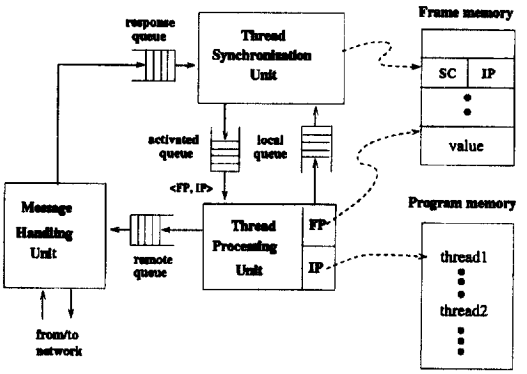
다음에는 논문의 전개 순서에 대해서 언급한다. 2장에서는 논문의 실행 모델에 대해서 설명하고, 3장에서는 다중스레딩에 의한 루프의 실행 방식을 설명하고, 루프의 실행 부담을 최소화할 수 있는 여러 가지의 방법들을 제안한다. 4장에서는 제안된 방법들에 대해서 시뮬레이션을 통한 분석을 수행하고, 5장에서는 결론을 언급한다.

2. 실행 모델

논문의 실행 모델은 데이터플로우와 폰 노이만 계산 모델의 혼합형으로 다중스레딩 실행 모델이다. 즉, 스레드간에는 데이터플로우 계산모델을 적용하여 자연스러운 동기화와 대규모 병렬성을 이용하고, 스레드 계산시에는 폰 노이만 계산 방식에 따라 캐쉬와 프로세서 레지스터를 통한 계산의 지역성을 이용함으로써 두 계산 모델의 취약점을 상호 보완한다. 프로그램은 코드 블록의 집합으로 표현되고, 각 코드 블록은 상관된 스레드의 집합으로, 각 스레드는 순차적으로 실행되는 명령어의 집합으로 계층적으로 구성된다. 여기서 코드 블록은 프로그램의 함수나 루프에 해당된다. DAVRID 실행 모델[3]에서 각 프로세서는 자신의 프로그램 메모리와 프레임 메모리를 갖는데, 스레드 코드는 프로그램 메모리에 저장되고, 데이터는 프레임 메모리에 저장된다. 코드 블록의 각 호출시에 그 실행 환경으로 프레임 메모리내 프레임이 할당된다. 배열등 구조화 데이터는 전역 메모리인 I-구조(I-structure)[11]상에 저장된다. I-구조에는 단지 한 번 쓰기의 규칙이 적용된다.

(그림 1)은 DAVRID 실행 모델에 대한 추상기계이다. 프로세서는 3개의 실행 장치 TPU(Thread Processing Unit), TSU(Thread synchronization Unit), MHU(Message Handling Unit)로 구성된다. 각 장치는 자신의 지역 메모리를 가지며, 독립적으로 동작한다. TPU는 활성스레드 큐(activated queue)로부터 연속(continuation)을 가져와서 해당 스레드를 순차적으로 실행시킨다. 연속은 (fp, ip)로 구성되는데, fp는 프레임의 시작 주소를 나타내며, ip는 스레드의 첫 명령어에 대한 주소를 나타낸다. 스레드 실행시 필요한 데이터는 fp가 가리키는 프레임으로부터 가져온다. 또한, 원격 데이터에 대한 요청은 원격 큐(remote queue)를 통해서 MHU에 전달한다. TSU는 주로 반응 큐(response queue)나 지역 큐(local queue)로부터 토큰을 가져와서 스레드간의 동기화를 수행하고, 이의 결과로서 활성화되는 스레드에

대해서는 해당 연속을 구성하여 이를 활성스레드 큐에 넣는다. MHU는 주로 메시지 전달 기능을 담당한다. 전역 메모리 I-구조는 MHU 상에 위치한다.



(그림 1) DAVRID의 추상기계

DAVRID 실행 모델에서 스레드는 sc(synchronization counter)와 ip(instruction pointer)로 표현된다. sc는 스레드가 활성화되기 전에 도달해야 하는 값(value)이나 신호(signal)의 개수를 나타내며, ip는 스레드의 첫 명령어가 저장된 프로그램 메모리의 주소를 나타낸다. 값이나 신호가 도달할 때마다 sc의 값이 1만큼 감소되고, 이 값이 0이 될 때 ip가 가리키는 스레드가 활성화된다. 즉, 스레드 실행에 필요한 값이 도달하거나 동기화가 이루어진 후에 그 스레드는 활성화될 수 있다. 따라서 스레드는 일단 실행되면 실행이 완료될 때까지 중단되지 않는다.

3. 루프 실행의 최적화

KU-Loop[10]은 다중스레딩 방식에 의한 함수 언어 Id의 루프 펼침 기법이다. 이 기법은 함수 실행 방식에 기반하여 K개의 프레임을 새로이 할당, 설정하고, 이들 프레임상에서 K개의 반복문을 동시에 실행시킴으로써 실행 효과를 얻는다. 프레임의 설정은 반복문 실행시 필요한 모든 루프 상수들을 미리 이 곳에 저장시킴으로써 루프 실행 환경을 설정하는 것으로 이루어진다. 또한, 루프 실행시에 각 프레임은 현재의 반복문을 실행 완료하였을 때, 다음번째 반복문을 실행하기 전에 프레임을 다시 초기화시키는 것이 필요하다. 왜냐하면 반복문이 실행되면서 스레드의 동기화 계수기의 값은 감소될 것이며, 반복문이 실행 완료되었을 때는 그 동기화 계수기의 값이 모두 0인 상태에 있기 때문이다. 루프 실행 환경의 설정, 프레임의 초기화는 다중스레딩에 의한 루프의 실행을 위해서 필연적인 부담이며, 이러한 실행 부담은 예상한 바와는 다르게 다중스레딩의 효과를 감소시킬 수 있다. 여기서는 루프의 실행 부담을 최소화시킬 수 있는 3가지 방법을 제안한다.

3.1 루프 프레임 초기화의 분산

루프 프레임의 초기화는 현재 반복문이 실행 완료되었을 때, 반복문을 구성하는 모든 스레드들의 동기화 계수기를 원래 상태로 초기화하는 것으로 이루어진다. 현재의 DAVRID 컴파일러는 스레드의 동기화 계수기를 초기화시키는데 4개의 명령어가 수행된다. 따라서 반복문 수행시 m개의 스레드가 실행되고, 루프의 총 반복구문 수가 n개일 때, 루프의 수행이 완료될 때까지 4mn의 명령어가 프레임을 초기화시키는데 실행된다. 이것은 적지 않은 실행 부담을 초래할 수 있다. 여기서는 DAVRID의 3개의 노드 가운데서 프로그램 실행시 SU의 부하가 TUP의 부하보다 낮다는 사실[3]로부터 착안하여, 현재 TPU에서 처리되고 있는 프레임의 초기화 작업을 SU로 이전시킨다. 이것은 노드들간에 부하를 균형화시킴과 동시에 반복문 실행과 프레임 초기화를 중첩시킴으로써 프로그램 실행 속도를 향상시킬 수 있다.

현재 SU는 값이나 신호가 도달할 때마다 해당 스레드의 동기화 계수기의 값을 감소시키고, 이 값이 0이 되었을 때, 그 스레드의 연속을 활성 스레드 큐에 집어넣는다. TPU는 이 큐로부터 연속을 하나씩 꺼내와서 해당 스레드를 그 스레드의 실행이 완료될 때까지 실행시킨다. 프레임 초기화는 TPU에 의해서 이루어진다. 따라서 스레드의 동기화계수기 값을 감소시키는 일과 초기화시키는 일이 구분되어 SU와 TPU에서 각각 수행된다. 논문에서는 이러한 두 가지의 작업을 구분하지 않고 SU가 모두 수행하도록 한다. 즉, 이제 SU는 동기화계수기의 값을 감소시키고, 이 값이 0일 때 해당 스레드의 연속을 활성 스레드 큐에 집어 넣음과 동시에 추가로 그 스레드의 동기화계수기의 값을 초기화시킨다. 그러나 SU가 스레드의 동기화계수기를 초기화시킬 때 원래의 값을 알 필요가 있다. 따라서 이러한 값을 프레임 어딘가에 복사해 둘 필요가 있다. 또한, SU는 단지 반복문 수행에 참여하는 스레드에 대해서만 동기화계수기의 값을 초기화시킬 필요가 있다. 다른

스레드들은 루프 실행과 관련이 없으므로 이들을 종전처럼 처리하는 것이 바람직하다. 이를 위해서 동기화 명령어(이것은 이 명령어의 실행 결과로 SU에서 동기화계수기 값이 변경되어지는 모든 명령어를 총칭한다)를 구분하는 것이 필요하다: 루프 본체에 속하는 동기화 명령어와 그렇지 않는 동기화 명령어. 논문에서는 루프 본체에 속하는 동기화 명령어를 추가로 설계하고, 컴파일러가 루프 문에 대한 코드 생성시에 이를 반영하고, 루프 프레임 구조와 SU의 실행시간 시스템에 이를 반영한다. 다음은 각 수정 사항에 대해서 설명한다.

먼저, 루프 본체에 속하는 동기화 명령어를 다른 동기화 명령어와 구분하기 위해 새로운 PML(Parallel Machine Language) 명령어를 추가한다. 기존의 각 동기화 명령어는 루프 본체에 속할 경우에는 대응되는 새로운 동기화 명령어로 대체된다. 기존의 동기화 명령어에 대응하여 설계된 새로운 동기화 명령어는 기존의 동기화 명령어 앞에 'L'을 붙여서 표현한다. 가령, 원격 메모리에 위치한 값을 요청하는 동기화 명령어 ILOAD에 대해서 새롭게 설계된 상응한 명령어는 LILOAD이다. ILOAD와 LILOAD 간에 의미상의 차이는 없다.

다음은 프레임 구조 수정 사항에 관한 것이다. SU가 루프 실행에 참여하는 스레드의 동기화계수기의 초기값을 쉽게 접근하여 참조할 수 있도록, 동기화계수기의 초기값을 (sc, ip)가 저장되어 있는 프레임 위치의 바로 다음번째 위치에 복사해 둔다. 즉, (sc, ip)의 값이 p의 프레임 주소에 저장되어 있다면, 동기화계수기 sc는 (p+2)의 프레임 주소에 저장된다. 여기서 2는 2 바이트, 즉 DAVRID에서 워드 크기를 나타낸다.

새롭게 설계된 동기화 명령어에 대해서 SU 실행 루틴을 설계하는 것이 필요하다. 새로운 동기화 명령어 LSTART의 실행 루틴을 기존의 상응한 동기화 명령어 START의 실행 루틴과 비교하여 설명한다. START 명령어는 값을 전달하고, 이 값을 대기하고 있는 스레드의 동기화 계수기를 감소시킨다. START 명령어의 형식은 다음과 같다.

START *fp, offset, disp, value*

여기서 *fp*는 프레임 포인터를 나타내며, *offset, disp*는 *fp*의 프레임 시작 주소로부터의 이격 거리를 나타내는데, (*fp+disp*)에 값 *value*가 저장되며, (*fp+offset*)에는 목표 스레드에 대한 (*sc, ip*)가 저장되어 있다. 여기서 *offset, disp*는 컴파일러에 의해서 계산된 값이다. START 명령어의 실행 루틴은 다음과 같다.

```
store value in FM[fp+disp]
get sc from FM[fp+offset]
if (sc == 1)
    then {
        construct a continuation (fp, ip)
        put the continuation in the activated queue
    }
    else
        decrease sc by 1
```

즉, START는 FM[*fp+disp*]에 value를 저장하고, FM[*fp+offset*]로부터 sc를 가져온다. 이 값이 1이면, 연속 (fp, ip)를 생성해서 이를 활성스레드 큐에 전달하고, 그렇지 않으면 단지 sc의 값을 1만큼 감소시킨다. 다음은 LSTART의 실행 루틴이다.

```
store value in FM[fp+disp]
get sc from FM[fp+offset]
if (sc == 1)
then {
        construct a continuation (fp, ip)
        put the continuation in the activated queue
        get an initial value of sc from FM[fp+offset+2]
        store (sc, ip) in FM[fp+offset]
}
else
        decrease sc by 1
```

실행 의미는 START의 실행루틴과 거의 동일하다. 이 루틴은 sc의 값이 1일 때 연속을 생성해서 활성스레드 큐에 전달하는 것 외에도, 추가로 sc의 초기값을 FM[fp+offset+2]로부터 가져와서 FM[fp+offset]내 sc의 값을 초기화시킨다.

루프 본체에 속하는 동기화 명령어는 루프 외부의 동기화 명령어와 다르다. 따라서 컴파일러가 스레드 코드를 생성할 때 이러한 사실을 반영해야 한다. 또한, 프레임 초기화 작업을 SU로 이전시킨 효과로 프레임 초기화 코드가 삭제될 수 있다.

다음 두 행렬에 대해서 내적을 계산하는 Id 프로그램 innerx를 통해서 루프 프레임 초기화의 분산 효과를 설명한다.

```
n = 1000;
int A = Iarray(1,100);
int B = Iarray(1,100);
s = 0;
res = (for k ← 1 to n step 1 do
          next s = s + A[k]*B[k];
       finally s);
In res
```

(그림 2)는 innerx 프로그램의 코드 일부로서 *for*-루프에 대한 스레드 코드를 보여준다. 이것은 루프를 펼치지 않고 순차적으로 실행하도록 생성된 코드이며, 논문에서 제안된 최적화 방법이 적용되기 이전의 코드이다. 각 스레드 코드의 시작과 끝은 레이블 THREAD_*와 END로 구분된다(레이블은 ':'으로 끝난다). 첫 번째 스레드 THREAD_1을 생각해 보자. 이 레이블 뒤에 '#'이 오는데 이것은 뒤에 오는 문장이 설명문임을 나타낸다. "(sc =1, offset = 2)"의 설명문은 이 스레드의 동기화 계수기가 1이고 프레임 시작점으로부터 offset 32의 위치에서부터 이 스레드 코드가 저장된다는 것을 말해준다. 레지스터 $6에는 프레임 포인터 fp가 저장되어 있으며, 첫 번째 load 명령어는 fp로부터 offset 72에 위치한 루프제어변수 k의 값을 레지스터 $8에 저장한다. THREAD_1은 k와 루프 경계값(loop bound) n을 비교해서 k ≤ n이면 I-구조에의 참조를 통한 두 배열 A, B의 하한 값(low bound) 값을 요청한다. 이 두 값 모두 I-구조로부터 도달하면 THREAD_2가 활성화된다(이것이 THREAD_2의 sc 값이 2인 이유이다). 이 스레드는 배열 원소 A[k], B[k]에 대한 실제 주소값을 계산하고, ILOAD 명령어를 통해서 이러한 배열 원소의 값을 I-구조에 요청한다. A[k], B[k]의 값이 모두 도달하면 THREAD_3이 활성화된다. 이 스레드는 A[k], B[k]의 두 값을 프레임으로부터 가져와서 이들을 곱하여 루프제어변수 s의 값을 증가시키고, k의 값을 갱신한 후 프레임을 초기화시킨다. 다음에 STARTd 명령어를 통해서 THREAD_1을 활성화시킴으로써 루프의 다음번째 반복문의 실행을 시작한다.

(그림 3)은 (그림 2)의 코드에 대해서 프레임 초기화 분산 기법을 적용시켜서 다시 생성한 코드이다. 이들 코드간에 다음 두 가지의 차이점을 볼 수 있다. 첫째 THREAD_1과 THREAD_2에서 동기화 명령어 ARRAY_BOUND, ILOAD가 LARRAY_BOUND, LILOAD로 각각 변경되었고, 둘째 THREAD_3에서 프레임을 초기화시키는 코드가 삭제되었다.

```
THREAD_1: #(sc =1, offset = 32)
        lw      $8,72($6)       # get k
        lw      $9,60($6)       # get loop bound
        sle     $10,$8,$9       # compare k <= loop bound
        beqz    $10,ELSE
        lw      $8,52($6)       # get A
        ARRAY_BOUND $8,36,76    # request the low bound of A
        lw      $9,56($6)       # get B
        ARRAY_BOUND $9,36,84    # request the low bound of B
        li      $10,8           # computing a part of A[k]'s address
        li      $11,1           # (8 * 1 + 16 + A)
        mul     $10,$10,$11
        add     $10,$10,16
        add     $10,$8,$10
        sw      $10,92
        li      $10,8           # computing a part of B[k]'s address
        li      $11,1
        mul     $10,$10,$11
        add     $10,$10,16
        add     $10,$9,$10
        sw      $10,96
        b       END
ELSE:   lw      $8,64($6)
        HOST_OUT1 1, $8         # send the result to host
        HALT                    # terminate program
END:    NEXT
THREAD_2: #(sc = 2, offset = 36)
        lw      $8,72($6)       # get k
        lw      $9,76($6)       # get low bound(lb) of A
        sub     $9,$8,$9        # compute a real address of A[k]
        li      $10,8           # (8*1+16+A+(k-lb)*8)
        mul     $9,$9,$10
        lw      $10,92($6)
        add     $9,$9,$10
        ILOAD   $9,40,100       # requset a value of A[k]
        lw      $9,84($6)       # get low bound(lb) of B
        sub     $8,$8,$9
        li      $9,8
        mul     $8,$8,$9
        lw      $9,96($6)
        add     $8,$8,$9
        ILOAD   $8,40,104       # request a value of B[k]
        NEXT
THREAD_3: #(sc = 2, offset = 40)
        lw      $8,100($6)      # get A[k]
        lw      $9,104($6)      # get B[k]
        mul     $8,$8,$9
        lw      $9,64($6)       # get s
        add     $8,$8,$9
        sw      $8,64($6)       # s = s+A[k]*B[k]
        lw      $15,72($6)      # get k
        lw      $16,68($6)      # get step value
        add     $17,$15,$16
        sw      $17,72($6)      # k= k + step value
        lui     $8,0x0020       # reset the loop frame below
        li      $9,THREAD_2
        or      $10,$8,$9
        sw      $10,36($6)
        li      $9,THREAD_3
        or      $8,$8,$9
        sw      $8,40($6)
        lw      $8,4($6)
        STARTd  $8,THREAD_1     # activate THREAD_1
        NEXT
```

(그림 2) innerx에 대한 스레드 코드

```
THREAD_1: #(sc =1, off = 32)
        .....
        lw      $8,52($6)       # get A
        LARRAY_BOUND $8,36,76   # request the low bound of A
        lw      $9,56($6)       # get B
        LARRAY_BOUND $9,36,84   # request the low bound of B
        .....
END:    NEXT
THREAD_2: #(sc = 2, off = 36)
        .....
        LILOAD  $9,40,100       # requset a value of A[k]
        .....
        LILOAD  $8,40,104       # request a value of B[k]
        NEXT
THREAD_3: #(sc = 2, off = 40)
        lw      $8,100($6)      # get A[k]
        lw      $9,104($6)      # get B[k]
        mul     $8,$8,$9
        lw      $9,64($6)       # get s
        add     $8,$8,$9
        sw      $8,64($6)       # s = s+A[k]*B[k]
        lw      $15,72($6)      # get k
        lw      $16,68($6)      # get step value
        add     $17,$15,$16
        sw      $17,72($6)      # k= k + step value
        lw      $8,4($6)
        STARTd  $8,THREAD_1     # activate THREAD_1
        NEXT
```

(그림 3) 프레임 초기화 분산 기법이 적용된 코드

3.2 루프 상수의 통합

KU-Loop에서 루프 실행 환경 설정은 원래의 루프 상수는 물론, 프레임간의 체인(chain) 정보, 루프 호출자에의 복귀 정보, 루프 본체를 구성하는 스레드에 대한 정보, 루프 경계값, 루프제어변수의 증가 단위값 등을 (이들을 모두 루프 상수라 칭한다) 각 프레임에 전달함으로써 이루어진다. 루프 상수의 전달은 첫 번째 프레임에 대해서는 루프 호출자에 의해서 이루어지며, 다른 프레임에 대해서는 프레임 체인 상의 직전에 위치한 프레임에 의해서 이루어진다. 루프 상수의 개수는 대개 10개 이상이며, 이들 모두가 개별적인 메시지를 통해서 프레임에 전달되고 있다. 따라서 그 부담은 상당할 것이며, 중첩 루프에서 내곽 루프를 펼칠 경우에 그 부담은 더욱 심각해질 것이다. 가령, 내곽 루프에 속한 루프 상수의 개수가 10개이고, 외곽 루프의 총 반복구문 수가 n인 경우에, 내곽 루프를 펼칠 때 루프 실행 환경 설정을 위해 총 10Kn개의 메시지 생성 및 전달이 필요하게 된다.

루프 상수는 루프를 펼치는 시점에는 그 값이 모두 알려져 있다. 따라서 가능한 많은 루프 상수를 한 메시지에 통합하여 보낼 수 있다. 이것은 루프 상수 전달에 따른 수많은 메시지의 생성, 전달, 처리에 따른 부담을 효과적으로 제거한다. 루프 상수를 한 메시지에 통합하여 전달하기 위해서 프레임에 전달된 루프 상수가 연속적으로 저장될 수 있도록 프레임 구조에 대한 조정이 필요하고, 여러 개의 루프 상수를 통합하여 전달할 수 있는 새로운 PML 명령어를 설계하는 것이 필요하다.

현재 루프 상수는 STARTN의 명령어를 통해서 전달되며, n개의 루프 상수를 전달하기 위해서 다음과 같이 n개의 STARTN 명령어가 필요하다.

STARTN fp, dis_1, val_1, off_1
STARTN fp, dis_2, val_2, off_2
.....
STARTN fp, dis_n, val_n, off_n

STARTN 명령어의 수행을 통해서 값 val_i이 목표 프레임의 (fp+dis_i)에 저장된다. 이때 목표 프레임은 송신자 측에 신호를 보내어 이 값이 도달하였음을 알리고, 이 결과로 송신자 프레임의 off_i에 위치한 sc의 값이 감소된다. 논문에서는 STARTNb의 새로운 명령어를 설계하여 다음과 같이 n개의 루프 상수를 통합하여 전달토록 한다.

STARTNb fp, dis, off, val_1, val_2, ..., val_n

이 명령어는 val_1, val_2, ..., val_n을 (fp+dis)에 순서대로 저장하는데, val_i는 (fp+dis+2*i)에 저장된다. 여기서 통합되는 루프 상수의 크기는 모두 동일해야 함을 알 수 있다. 또한, 루프 상수의 통합을 통해서 단지 한 개의 신호가 송신자 측에 전달된다는 것을 알 수 있다.

3.3 배열 연산의 최적화

DAVRID에서 배열은 I-구조에 저장된다. 배열의 한 원소에 대한 참조 A[k]에 대한 실제 주소 RA는 다음과 같이 계산된다.

RA = A + es * dim + ds + (k - lb) * es

여기서 es는 배열 원소의 크기를, dim은 배열의 차원을, ds는 타입등 배열에 관한 정보가 저장된 배열 서술자(descriptor)의 크기를, lb는 배열 첨자의 하한 값을 각각 나타낸다. DAVRID에서 ds는 16이며, es는 정수형인 경우 8 바이트(2 워드)이다. 정수형 배열 원소가 2워

드에 저장되고 있는데, 이는 각 원소에 대해서 존재 비트로 한 워드가 포함되기 때문이다. 또한, 현재의 컴파일러는 각 배열 원소에 대한 주소 계산시에 ARRAY_BOUND 명령어를 통해서 lb 값을 얻는다. 이 명령어는 전역 메모리 I-구조로부터 값을 요청하기 때문에 긴 지체시간을 초래하는 명령어이다.

RA의 수식을 살펴보면, 루프제어변수 k를 제외하고는 상수로 처리될 수 있음을 알 수 있다. 따라서 이러한 상수항으로 구성된 공통 부분 수식을 미리 계산해 두고 이를 공유함으로써 배열 연산을 최적화시킬 수 있다. 다음은 RA의 수식을 다시 작성한 것이다.

$$RA' = (A + es*dim + ds - lb*es) + (k*es)$$

왼쪽 항은 상수항들로 구성된 공통 부분 수식이다. 따라서 이 부분 수식을 미리 계산해 놓고서, 루프 실행시 배열 원소에 대한 주소 계산은 오른쪽 항만을 계산하고, 여기에 미리 계산된 부분 수식의 값만을 더하면 된다. 여기서 중요한 사항은 부분 수식에 lb의 항이 포함되어 있다는 것이다. 따라서 동일한 배열에 대해서 각 원소의 주소 계산은 처음의 ARRAY_BOUND를 통해서 가져온 lb를 공유할 수 있다.

공통 부분 수식의 공유에 따른 루프 실행의 효과를 살펴보기 위해, 루프의 총 반복구문수가 n이고 그 본체에서 3개의 배열을 참조하는 루프를 생각해 보자. 공통 부분 수식이 공유되지 않은 경우에는, 각 반복문 실행시 배열 원소 참조를 위해서 3개의 ARRAY_BOUND 명령어가 실행될 것이고, 따라서 루프가 실행 완료될 때까지는 총 3n개의 ARRAY_BOUND 명령어가 실행될 것이다. 그러나 공통 부분 수식이 공유될 경우에는 루프 실행 전에 단지 3개의 ARRAY_BOUND 명령어만이 실행될 것이다. 배열 원소 참조가 동일한 배열에 대해서 이루어질 경우에는 단지 한 개의 ARRAY_BOUND 명령어 만이 수행될 것이다.

(그림 4)는 (그림 3)의 코드에 대해서 배열 연산 최적화가 적용된 코드이다. 배열 원소에 대한 주소 계산시 공통 부수식의 공유로 (그림 3)의 THREAD_1내 LARRAY_BOUND 명령어가 제거될 수 있고, 이로써 THREAD_1과 THREAD_2가 한 개의 스레드로 합병되었다. 두 배열 A, B에 대한 lb는 이 루프가 실행되기 전에 미리 계산되어 프레임 슬롯 76, 80에 각각 저장되었음을 볼 수 있다.

```
THREAD_1: #(sc =1, offset = 32)
        lw      $8,72($6)       # get k
        lw      $9,60($6)       # get loop bound
        sle     $10,$8,$9       # compare k <= loop bound
        beqz    $10,ELSE
        li      $10,8           # load an immediate value es
        mul     $10,$8,$10      # compute k*es
        lw      $8,76($6)       # get A'(= A+es*dim+ds-lb*es)
        add     $8,$8,$10       # compute A' + (k*es)
        LILOAD  $8,40,100       # request a value of A[k]
        lw      $8,80($6)       # get B'
        add     $8,$8,$10       # compute B'+(k*es)
        LILOAD  $8,40,104       # request a value of B[k]
        b       END
ELSE:   lw $8,64($6)
        HOST_OUT1 1,$8          # send the result to host
        HALT                    # terminate program
END:    NEXT

THREAD_2: #(sc = 2, offset = 40)
        lw      $8,100($6)      # get A[k]
        lw      $9,104($6)      # get B[k]
        mul     $8,$8,$9
        lw      $9,64($6)       # get s
        add     $8,$8,$9
        sw      $8,64($6)       # s = s+A[k]*B[k]
        lw      $15,72($6)      # get k
        lw      $16,68($6)      # get step value
        add     $17,$15,$16
        sw      $17,72($6)      # k= k + step value
        lw      $8,4($6)
        STARTd  $8,THREAD_1     # activate THREAD_1
        NEXT
```

(그림 4) 배열 연산 최적화가 적용된 코드

4. 시뮬레이션

이 논문에서 제안된 루프의 최적화 기법을 다중 스레드 구조 DAVRID 상에서 시뮬레이션을 통해서 그 효과를 분석한다. 사용된 벤치마크는 행렬 곱셈 프로그램이다. 이 프로그램은 3의 중첩 수준을 갖는 루프를 포함한다. 각 루프는 4만큼씩 펼쳐져서 수행되었으며, 프로그램 실행은 16개의 노드를 갖는 DAVRID 시뮬레이터 상에서 이루어졌다. 논문에서 제안한 프레임 초기화 분산, 루프 상수 통합, 배열 연산 최적화 등의 루프 최적화 기법은 현재 컴파일러에 구현되지 않은 상태이다. 따라서 프로그램 실행에 사용된 최적화 코드는 컴파일러로부터 생성된 코드를 손으로 작업하여 생성하였다.

이 표는 프로그램 실행시 처리된 명령어들의 빈도수를 명령어 종류별로 보여준다. 이 표는 3가지의 루프 최적화 기법을 적용하기 전과 적용한 후의 결과를 비교한다.

〈표 1〉 명령어들의 빈도수

구분 \ 종류	Synch	Memory	SM	ALU	Load/Store	Branch	Others
최적화 전	132,208	2,108	32,800	310,131	528,012	39,072	366,582
최적화 후	123,636	2,108	16,403	235,174	350,419	39,072	203,803
비율	1.07	1.00	2.00	1.32	1.51	1.00	1.80

위에서 Synch는 동기화를 수행하는 명령어들의 그룹이며, Memory는 프레임이나 힙 메모리의 할당과 반환을 수행하는 명령어들의 그룹이며, SM은 구조화 메모리에 접근하는 명령어들의 그룹이며, ALU는 산술, 논리 연산을 수행하는 명령어들의 그룹이다. Load/Store는 스레드 경계선 상에서 수행되는 프레임으로부터의 값의 적재와 저장을 수행하는 명령어들의 그룹이며, Branch는 제어의 분기를 표현하는 명령어들의 그룹이며, Others는 스레드가 종료되기 직전에 수행되는 명령어, 프레임을 초기화하거나 재설정하는데 사용되는 명령어들의 그룹이다.

Synch 명령어의 감소는 루프 상수의 통합 결과이다. 왜냐하면 각 루프 상수 전달은 동기화 명령어로 표현되고, 이러한 루프 상수들의 통합으로 여러 개의 동기화 명령어가 한 개의 동기화 명령어로 통합되어 표현될 수 있었기 때문이다. SM 명령어의 감소는 배열 연산 최적화에 따른 결과이다. 행렬 곱셈은 배열 원소들을 빈번히 참조한다. 최적화 전의 코드에서는 각 배열 원소 참조에 대해서 구조화 메모리에의 접근을 통해서 배열의 경계 값을 참조하였다. 그러나 배열 연산 최적화의 결과로 동일한 배열의 경계 값을 그 배열의 모든 원소 참조에서 공유할 수 있게 함으로써 SM 메모리에의 접근이 감소되었다. ALU 명령어의 감소도 배열 연산 최적화의 결과로 해석된다. 왜냐하면 배열 원소 참조시에 그 원소의 주소를 계산하는 식이 동일한 배열 원소 참조간에 공유될 수 있었기 때문이다. Load/Store 명령어의 감소는 최적화의 결과로 스레드의 개수가 감소하였기 때문이다. 따라서 스레드 실행시 프레임으로부터 값을 적재하는 Load 명령어와 스레드 종료시에 값을 프레임에 저장하는 Store 명령어가 감소하였다. 실제로, 루프를 구성하는 스레드의 개수가 최적화 전에는 41개였으나 최적화의 결과로 38개로 감소하였다. 스레드는 가운데 위치한 루프에서 1개, 최내곽에 위치한 루프에서 2개가 각각 감소하였다. 스레드의 감소는 배열 연산의 최적화로 ARRAY_BOUND 명령어의 감소 결과이다. Others 명령어의 감소는 루프 실행부담의 분산 결과이다. 즉, 매 반복시에 프레임을 초기화시킬 필요를 제거한 결과이다. 마지막으로, 메모리 할당과 반환에 관한 명령어와 제어 분기에 관한 명령어는 최적화에 영향을 받지 않음을 알 수 있다.

〈표 2〉 DAVRID 노드의 효율성

구분 \ 장치	TPU	SU	MHU
최적화 전	42.63	33.25	99.96
최적화 후	48.60	43.70	99.91

<표 2>는 DAVRID 구조의 각 장치에 대한 효율성을 최적화 전과 후에 대해서 비교한다. SU의 효율성(utilization)이 최적화의 결과로 약 10% 정도 향상되었음을 알 수 있다. 이것은 루프 실행 부담의 분산 결과이다. 즉, SU가 동기화 명령어 처리시에 프레임 재설정을 부분적으로 추가로 처리한다. 그러나 TPU의 효율성도 약간 향상되고 있는데, 이것은 루프 상수의 통합과 배열 연산 최적화의 결과로 해석된다. 이들의 최적화로 긴 지체시간을 갖는 명령어가 감소되었으며, 이러한 명령어의 처리 결과를 대기하는 시간이 감소하였기 때문이다. MHU는 최적화 전과 후 모두 포화 상태에 있음을 알 수 있다. 이것은 4개의 노드간에 공유되는 MHU에 병목 현상이 발생하였기 때문이다. 배열을 저장하는 구조화 메모리가 MHU에 위치하고 있으며, 행렬 곱셈은 배열 연산을 상당히 포함하고 있기 때문에, 노드들의 공유된 MHU에 대한 접근이 상당하였던 것으로 해석된다.

프로그램 실행 속도에 대한 비교는 다음과 같다. 최적화 전에는 4,164,282의 클럭 사이클이 소요되었으며, 최적화 후에는 2,319,545의 클럭 사이클이 소요되었다. 따라서 최적화에 따른 속도 향상은 1.8이다. 여기서 한 클럭 사이클은 ALU 명령어의 처리 시간을 나타낸다. MHU 상의 병목 현상을 해결한다면 속도가 더욱 향상될 것으로 본다.

5. 결 론

본 논문에서는 다중스레딩 실행 환경에서 루프의 실행을 최적화시킬 수 있는 방법으로 루프 실행 부담의 분산, 루프 상수의 통합 전달, 배열 연산의 최적화 등을 제안하고, 행렬 곱셈 프로그램에 대한 시뮬레이션을 통해서 그 효과를 분석하였다. 시뮬레이션의 결과는 루프 실행 부담의 분산 결과로 SU의 효율성이 향상되었으며, 루프 상수의 통합 전달과 배열 연산 최적화의 결과로 해당 명령어의 실행 빈도수가 감소하였음을 보여준다. 그러나 4개의 노드간에 공유되는 MHU 상에 병목 현상이 초래되어 속도 향상에 제한이 있음을 보여준다. 앞으로 이러한 병목 현상을 제거할 수 있는 기계 구조적 혹은 소프트웨어적인 방법이 연구되어야 할 것으로 본다. 또한, 본 논문에서 제안된 방법이 다양한 벤치마크 상에서 분석될 수 있도록 컴파일러 상에 구현되어야 한다.

참 고 문 헌

[1] Arvind, R. A. Iannucci, Two Fundamental Issues in Multiprocessing, CSG Memo 226-5, Lab. for Computer Science, MIT, Cambridge, MA 02139, 1986.

[2] D. E. Culler, S. C. Goldstein, et al., TAM-A Compiler Controlled Threaded Abstract Machine, Journal of Parallel and Distributed Computing, Vol.18, pp.347-370, July 1993.

[3] S. Ha, J. Kim, et al., A Massively Parallel Multithreaded Architecture: DAVRID, In {\it Proc. of Int. Conf. on Computer Design, pp.70-74, Oct. 1994.

[4] B. S. Ang, D. Chiou, et al, Message Passing Support on StarT-Voyager, CSG Memo 387, MIT Lab. for Comp. Sci., July 16, 1996.

[5] R. S. Nikhil, Id Reference Manual, Version 90.1, CSG memo 284-2, MIT Lab for Comp. Sci., Sept. 1990.

[6] K. E. Schauser, D. E. Culler, T. Eicken, Compiler-Controlled Multithreading for Lenient Parallel Languages, In Proc. of the 5th FPCA, pp.50-72, 1991.

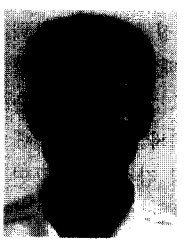
[7] K. R. Traub, D. E. Culler, and K. E. Schauser, Global Analysis for Partitioning Non-Strict Programs into Sequential Threads, In Proc. of ACM Conf. on LISP and Functional Programming, pp. 324-334, 1992.

[8] S. Ha, S. Han, and H. Kim, Partitioning a Lenient Parallel Language into Sequential Threads, In Proc. of the 28th Hawaii Int. Conf. on System Sciences, Vol.2, pp.83-92, 1995.

[9] S. Ha, H. Kim, and S. Han, The Efficient Implementation of Sequential Loops in Multithreaded Computation, In Proc. of the 7th IASTED-ISMM Int'l Conf. on Parallel and Distributed Computing and Systems, 1995.

[10] S. Ha and H. Kim, Loop Unfolding and Its Degree Determination for Multithreaded Computation, In Proc. of the 11th Annual International Symposium on High Performance Computing Systems, 1997.

[11] Arvind and R. S. Nikhil, I-Structures : Data Structures for Parallel Computing, Lecture Notes on Computer Science, Vol.279, pp.336-369, Oct. 1986.



하 상 호

e-mail : hsh@ai-cse.sch.ac.kr

1988년 서울대학교 자연과학대학 계산통계학과 졸업(학사)

1991년 서울대학교 자연과학대학 계산통계학과 졸업(석사)

1995년 서울대학교 자연과학대학 전산과학과 졸업(박사)

1995년~1996년 한국전자통신연구원 PostDoc.

1996년~1997년 미국 MIT PostDoc.

1997년~현재 순천향대학교 정보기술공학부 조교수

관심분야 : 프로그래밍 언어, 병렬/분산 처리 등