다중스레드 구조를 위한 함수형 언어의 중첩루프 펼침

(Unfolding Nested Loops of Functional Languages for Multithreaded Architectures)

하 상 호 †

(Sangho Ha)

요 약 Id 언어와 같은 함수형 언어의 중첩루프에 포함된 미세한 수준의 대규모 병렬성을 다중스레드 구조상에서 이용하려면 프로세서뿐만 아니라, 이름공간을 위한 상당히 많은 기억공간 등의 자원이 추가로 요구된다. 이러한 병렬성을 포함하는 중첩루프를 시스템 자원 제한 없이 무분별하게 펼쳐서 실행하려고 한다면, 실행도중 기억공간의 자원의 고갈로 인하여 프로그램의 실행이 중단될 수 있다. 또한, 루프의 펼침에 따른 부담으로 인하여 프로세서의 수에 비해서 루프를 지나치게 많이 펼치는 경우에, 병렬 수행의 효과가 상당히 떨어질 수 있다. 본 논문에서는 함수형 언어의 중첩루프를 다중스레드 구조 상에서 효과적으로 펼쳐서 실행할 수 있는 알고리즘을 제안하고 분석한다. 제안된 알고리즘의 특성은 주어진 중첩루프를 펼칠 시점에 프로세서 수와 기억공간의 현재 사용 가능한 시스템 자원 양에 제한하여 안전하면서도 가능한 최적으로 펼친다는데 있다.

키워드 : 다중스레딩, 다중스레드 구조, 함수형 언어, 중첩루프, 루프 펼침

Abstract We need an enormous amount of memories for name spaces as well as additional processors if we are to effectively exploit a massively parallelism in nested loops of functional languages such as Id. If there is no sufficient amount of memories enough to exploit that parallelism, the execution of programs can be aborted during the unfolding of loops. Additionally, if loops are overunfolded, compared with the number of processors available, the system performance can be degraded severely due to the overhead of loop unfolding. This paper suggests and analyzes an algorithm which can be used to effectively unfold nested loops of functional languages on multithreaded architectures. This algorithm has a feature to unfold a given nested loop safely and near optimally, considering the system resources of processors and memories available when the loop is to be unfolded.

Key words : Multithreading, Multithreaded Architectures, Functional Languages, Nested Loops, Loop Unfolding

1. 서 론

VLSI의 전자공학기술의 발달에 힘입어 병렬 컴퓨터가 1960년대부터 개발되어 오고 있지만 메모리 참조와 동기화로 인하여 확장성에 커다란 문제[1]를 안고 있다. 다중스레드 구조(multithreaded architecture)는 통신/동기화의 수행과 계산(computation) 수행을 효과적으로 중첩시킴으로써 이와 같은 문제를 해결한다. 즉, 원격 메모리에 대한 참조와 같이 긴 지체시간을 초래하는 명령어를 요청(request)과 반응(response)의 두 트랜잭션으로 분할하여 처리함으로써 반응 대기에 따른 프로세서의 유휴시간을 제거함으로써 실행의 효과를 높인다. 다중스레드 구조에서 실행 단위는 순차적으로 실행되는 명령어의 집합으로 정의되는 스레드(thread)이며, 이는 일반적으로 프로세스(process)에 비해서 그 크기가 작으며, 상태를 유지하기 위한 정보도 적어 문맥 교환에 따른 부담도 적

· 이 논문은 2000년도 한국학술진흥재단의 지원에 의하여 연구되었음 (KRF-2000-003-E00252)

† 종신회원 : 순천향대학교 정보기술공학부 교수
hsh@sch.ac.kr

논문접수 : 2002년 3월 2일
심사완료 : 2002년 8월 26일

다. 그러나 스레드간의 문맥교환이 빈번히 발생할 수 있으므로 다중스레드 구조는 스레드의 효과적인 실행을 위해서 빠른 문맥 교환을 수행할 수 있는 하드웨어적인 기능을 제공한다[2]. 최근에 개발되었거나 개발중에 있는 다중스레드 구조에는 *T[3], TAM[4], EM-X[5], DAVRID[6], StarT-Voyager[7] 등이 있다.

위의 다중스레드 구조에서 프로그램이 효과적으로 실행되기 위해서는 프로그램이 병렬성을 충분히 포함하는 것이 필요하다. 이는 실행중인 스레드가 원격 메모리 참조나 동기화의 수행으로 인하여 실행이 중단될 때, 이미 활성화된 다른 스레드가 존재하지 않는다면 프로세서는 여전히 유휴상태에 있기 때문이다. 기존의 명령형 언어는 암시적으로 기억 장소의 상태를 나타내기 때문에 기본적으로 순차적 실행을 표현하며, 따라서 병렬성을 표현하는데는 매우 제약적이다. 또한, 이와 같은 순차적 프로그램을 컴파일러가 병렬실행 가능한 코드로 변환시키는데도 한계가 따른다. 기존의 대부분 병렬 컴파일러는 별칭(alias)을 갖는 변수나 포인터 변수를 포함하는 코드에 대해서는 병렬화의 대상에서 아예 제외하고 있다. 이에 반해서, 함수형 언어에서는 상태의 개념이 표현되지 않기 때문에 미세한 수준의 병렬성이 자연스럽게 표현될 수 있다. 더욱이, Id[8]나 pH[9]의 함수형 언어에서와 같이 조기 계산(eager evaluation) 방식을 갖고 non-strict 의미론을 갖는 함수형 언어는 미세한 수준의 대규모 병렬성을 표현할 수 있다. 또한, 함수형 언어에서 프로그램은 선언적으로 표현되기 때문에 명령형 언어에 비해서 프로그램을 간명하고 추상적으로 표현할 수 있고, 함수형 언어가 수학적 기반을 두고 있어 프로그램 개발의 체계적인 분석과 접근이 가능하여 차세대 언어로서 그 중요성이 부각되고 있다.

최근에 조기 계산과 non-strict 의미론을 갖는 비정형성 함수형 언어가 갖는 미세한 수준의 대규모 병렬성을 다중스레드 구조상에 효과적으로 사상(mapping)시키기 위한 연구가 이루어져 왔는데[10,11,12,13], 이러한 연구의 주요 목적은 함수형 언어로부터 효율적이면서도 실행의 교착상태(deadlock)를 유발하지 않는 다중스레드 코드를 생성하는데 있으며, 따라서 스레드 실행 길이(run length)(이는 스레드의 실행 개시후 중단될 때까지의 실행시간으로 정의된다)의 최대화, 스레드간의 통신 및 동기화 부담의 최소화, 스레드간의 문맥교환의 최소화, 계산의 지역성 이용의 최대화 등을 통한 효율적인 스레드 코드의 생성과 스케쥴링에 그 주안점이 주어졌다.

루프(loop)는 프로그램 상에서 대부분의 병렬성을 내포하고 있으며, 따라서 Fortran이나 C의 프로그램을 병렬화시키는 병렬 컴파일러 영역에서 루프는 병렬화의 대상으로 매우 중요하게 다루어져 왔다. 비정형성 함수형 언어에서도 리커젼이나 루프의 반복구문이 상대적으로 많은 병렬성을 포함하고 있으며, [14]에서는 Id 프로그램에 포함된 루프를 다중스레드 구조 상에 효과적으로 구현하는 방법을 제시하고 있다. 이 방법은 루프의 반복간에 존재하는 종속 여부에 따라서 루프를 순차 루프와 병렬 루프로 구분하고, 각 루프 종류의 특성과 언어의 조기 계산 방식과 non-strictness를 반영하여 루프를 최적으로 펼칠 수 있는 알고리즘을 제시하고 있다. 그러나 이 방법은 단일 루프에 그 적용이 제한되며, 중첩루프를 고려하지 않는다.

프로그램에 흔히 포함되어 있는 중첩루프는 대부분 상당히 많은 병렬성을 포함하고 있으며, 이러한 중첩루프를 효과적으로 펼쳐서 실행하는 것이 병렬 수행에 매우 중요하다. 특히, Id 언어와 같은 비정형성 함수형 언어의 중첩루프에 포함된 미세한 수준의 대규모 병렬성을 이용하는데는 프로세서뿐만 아니라, 이름공간을 위한 상당히 많은 기억공간 등의 자원이 추가로 요구된다. 이러한 병렬성을 포함하는 중첩루프를 시스템 자원 제한없이 무분별하게 펼쳐서 실행하려고 한다면, 실행도중 기억공간의 자원의 고갈로 인하여 프로그램의 실행이 중단될 수 있다. 또한, [14]에서 지적하고 있듯이, 루프의 펼침에는 펼침 부담이 반드시 따른다. 루프를 펼치는 시점에 사용 가능한 프로세서의 수에 기반해서 루프를 펼치는 것이 중요하다. 프로세서의 수에 비해서 루프를 지나치게 많이 펼치는 경우에, 루프에 포함된 병렬성을 제대로 이용하지 못하며, 루프의 펼침 부담으로 인하여 병렬 수행의 효과가 상당히 떨어질 수 있고 심각한 경우에는 역효과가 초래될 수 있기 때문이다.

따라서 이러한 사실들을 반영하여 중첩루프를 효과적으로 펼쳐서 수행할 수 있는 방안이 요구된다. 그러나 다중스레드 구조 상에서 비정형성 함수형 언어의 중첩루프를 효과적으로 펼치는 연구는 아직 수행되지 않았다. 본 논문에서는 다중스레드 구조 상에서 비정형성 함수형 언어의 중첩루프를 효과적으로 펼쳐서 실행할 수 있는 알고리즘을 제안한다. 제안된 알고리즘의 특성은 주어진 중첩루프를 펼칠 시점에 프로세서 수와 기억공간의 현재 사용 가능한 시스템 자원 양에 제한하여 가능한 최적으로 펼친다는데 있다.

관련 연구로, Polychronopoulos가 제안한 Fortran이나 C의 프로그램에 내포된 중첩루프에 대해서 주어진 프로세서 개수를 최적으로 할당하는 OPTAL 알고리즘[15]을 들 수 있다. 그러나 이 알고리즘은 시스템의 자

원으로 단지 프로세서만을 고려하고 있다. 또한, 이 알고리즘은 공유메모리를 갖는 다중프로세서와 미세 수준의 병렬성 표현이 제한된 Fortran이나 C의 순차적 프로그램 환경을 고려하고 있는데, 이러한 알고리즘은 논문이 고려하고 있는 실행 환경, 즉 공유 메모리를 갖지 않는 다중스레드 구조나 미세한 수준의 대규모 병렬성이 내포된 비정형성 함수형 언어의 프로그램 환경에는 적합하지 않다. 또한, Loop Coalescing[15]을 통해서 중첩루프를 단일 루프로 변환하여, 단일 루프의 펼침 기법을 적용할 수도 있겠으나, 논문이 고려하고 있는 완전 중첩루프(perfectly nested loop)(각 중첩 수준에 단지 한 개의 루프만이 존재한다)에 적용하기 위해서 루프의 반복간에 존재하는 종속 방향(dependence direction)[16]이 모두 '='이어야 하는 제약사항이 따른다. 논문의 방법은 이러한 제약없이 적용 가능하다.

논문의 순서는 다음과 같다. 먼저, 2장에서는 논문의 실행 모델을 간단히 설명하고, 3장에서는 다중스레드 구조에서 비정형성 함수형 언어의 단일 루프 펼침 방법을 설명하고, 4장에서는 다중스레드 구조에서 비정형성 함수형 언어의 중첩루프를 효과적으로 펼칠 수 있는 중첩루프 펼침 알고리즘을 제안하고 분석한다. 마지막으로, 5장에서 결론을 맺는다.

2. 실행 모델

논문의 실행 모델은 데이타플로우와 폰 노이만 계산 모델의 혼합형으로 다중스레딩 실행 모델이다. 즉, 스레드간에는 데이타플로우 계산모델을 적용하여 자연스러운 동기화와 대규모 병렬성을 이용하고, 스레드 내부 계산시에는 폰 논이만 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. 이러한 혼합형 계산 모델의 하나인 DAVRID 실행 모델[6]에서, 각 프로세서는 자신의 프로그램 메모리와 프레임 메모리를 갖는데, 스레드 코드는 프로그램 메모리에 저장되고, 데이타는 프레임 메모리에 저장된다. 코드 블록의 각 호출시에 그 실행 환경으로 프레임 메모리로부터 해당 프레임이 할당된다. 배열 등 구조화 데이타는 전역 메모리인 I-구조(I-structure)[8] 상에 저장된다. I-구조에는 단지 한번 쓰기의 규칙이 적용된다.

논문이 고려하고 있는 DAVRID 실행 모델에서, 스레드는 동기화 계수기를 나타내는 sc와 명령어 포인터인 ip로 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. 따라서 스레드 실행에 필요한 값이 도달하거나 동기화가 이루어진 후에 스레드는 활성화될 수 있으며, 이 경우 스레드는 실행이 완료될 때까지 중단되지 않고 실행될 수 있다.

3. 단일 루프 펼침

KU-Loop[14]은 다중스레딩 방식으로 비정형 함수형 언어 Id의 루프를 펼치는 기법이다. 이 기법은 함수 실행 방식에 기반하여 루프를 펼친다. 프로그램상의 펼침 대상 루프는 루프 호출자와 루프 피호출자로 구분되어 번역된다[13]. 프로그램 실행시 루프가 실행될 시점에, 루프 호출자가 루프 피호출자를 호출하게 되고, 루프 피호출자는 루프를 펼쳐서 실행하고, 그 실행이 종료되면 루프 실행 결과를 루프 호출자에 반환한다. KU-Loop은 loop-setup, loop-exec, loop-cleanup의 세 부분으로 구성된다. loop-setup은 루프 병렬 실행을 위해서 K개의 프레임을 할당하고, 할당된 프레임을 루프 상수들로 미리 설정하며, loop-exec은 K개의 프레임상에서 K개의 반복을 병렬로 실행하며, loop-cleanup은 루프 실행 종료후에 루프 펼침을 위해서 할당되었던 모든 프레임을 시스템에 반환한다. loop-setup과 loop-exec에 의한 루프 병렬 실행 환경 설정과 루프 병렬 실행의 방법은 루프의 특성, 즉 순차 루프와 병렬 루프에 따라서 다르다. 순차 루프는 반복간에 자료 종속이 존재하는 루프이며, 병렬 루프는 그러한 자료 종속이 존재하지 않아서 각 반복이 독립적으로 실행 가능한 루프를 말한다. 루프 특성에 관계없이, KU-Loop을 구성하는 loop-setup, loop-exec, loop-cleanup은 모두 스레드들로 구성된다. 스레드는 2장의 실행 모델에서 언급하였듯이, 한번 실행되면 중단없이 종료시까지 계속 실행되며, 따라서 그 코드를 실행하는데 필요한 데이타가 모두 도달한 경우에 실행이 시작될 수 있다. 스레드의 동기화 계수기는 이 스레드 실행에 필요한 외부로부터 전달되는 데이타나 신호의 개수를 나타낸다.

루프 $L = (N, B)$를 생각해 보자. 여기서 N은 L의 반복구문수이고, B는 L의 본체로 한 반복구문의 계산시간을 나타낸다. L을 k개의 프로세서 상에 펼쳐서 실행시켰을 때의 수행시간을 $T_k(L)$이라 하면, $T_k(L)$은 다음

과 같이 표현될 수 있다:

$$T_k(L) = NB, \quad k = 1$$
$$NB/k + k\sigma, \quad k \geq 2 \qquad (1)$$

여기서 σ는 루프를 펼쳐서 실행하는데 따르는 부담으로, 주로 한 개의 루프 프레임을 설정하고 반환하는데 걸리는 시간을 나타낸다. NB는 L의 총 계산시간을 나타내고, k개의 반복구문이 병렬 수행되므로 L의 병렬 수행시간은 NB/k일 것이다. 그러나 여기에 루프 펼침 부담이 추가되어야 한다. 루프 프레임은 루프 펼침도 k만큼 설정되므로, 루프 펼침에 따른 부담은 $k\sigma$로 계산된다. 여기서 루프 펼침도(loop unfolding degree)는 루프를 펼치는 정도를 나타내는 것으로서, 루프 펼침도가 i이면 i개의 반복구문수가 동시에 실행될 수 있다는 것을 의미한다.

주어진 단일 루프에 대해서 루프 펼침도 k를 결정하는 식은 (1)의 식을 최소화할 수 있는 k의 값을 결정하는 것이며, 이 방법은 순차 루프와 병렬 루프의 루프 종류에 따라서 다르게 고려된다. 다음 절에서는 루프의 각 종류에 대해서 k를 결정하는 식에 대해서 살펴본다.

3.1. 순차 루프 펼침

순차 루프는 반복간에 자료 종속이 존재한다. 따라서 순차 루프의 병렬화로 여러 개의 반복이 동시에 실행되는데, 반복간의 종속에 따른 데이타 통신과 동기화가 요구된다. 순차 루프의 이러한 특성 때문에, 루프가 실행되기 전에 루프 병렬 실행 환경이 먼저 설정되어야 한다. 즉, loop-setup 과정이 loop-exec가 실행되기 전에 종료되어야 한다. 이것은 루프가 병렬 실행되기 전에 K개의 프레임 간에 체인이 형성되어야 하기 때문이다. 프레임간의 체인은 인접 프레임간에 통신하는데 필요한 정보이며, 이러한 정보를 이용하여 필요에 따라 데이타나 제어 신호를 인접 프레임에 전달한다.

loop-setup의 종료로, 루프 병렬실행 환경 설정이 완료되면, loop-exec의 실행으로 루프의 여러 개의 반복(iteration)이 각 프레임 상에서 동시에 실행된다. 각 반복은 루프 변수들의 값을 갱신시키는데, 이러한 값들은 다음 번째 반복에서 사용될 것이고, 이 다음 번째 반복 실행은 현재 반복이 프레임 $i(1 \leq i \leq K)$에서 실행되고 있다면 그 인접 프레임 j $(= (i+1) \bmod K)$에서 실행될 것이다. 프레임 j는 프레임 i로부터 다음 번째 반복 실행을 위한 루프 변수들의 값을 받아들이기 전에, 현재 수행하고 있는 반복을 실행 완료해야 한다. 그렇지 않으면, 잘못된 값을 갖는 루프 변수를 참조함으로써 프로그램 결과가 올바르지 않을 수 있다. 가령, 프레임 j는 현재 반복을 실행하고 있고, 프레임 a의 주소로부터 루프 변수 A의 값을 읽어들인다고 생각하자. 프레임 j가 프레임 a 주소로부터 값을 읽어들이기 전에, 프레임 i가 A의 값을 갱신하고, 이 값을 프레임 j에 보내어 a의 주소에 저장할 경우에, 프레임 j는 나중에 잘못된 A의 값을 참조하게 될 것이다. 이 경우 프레임 j가 참조하게 되는 A의 값은 현재 반복 $l(1 \leq l \leq n-K,$ n은 루프 경계) 이 아닌 다음 번째 반복 $(l+K)$에서 사용되어야 할 것이다. 이러한 루프 변수에 대한 잘못된 참조를 방지하기 위해서 프레임간에 동기화가 필요하다. 즉, 프레임 i는 프레임 j에게 갱신된 루프 변수를 전달하기 전에, 프레임 j가 현재 반복 수행을 종료하였고, 다음 번째 반복을 수행할 준비가 되어 있음을 확신해야 한다. 이러한 프레임간의 동기화를 위해서, 프레임 j는 현재 반복 수행을 종료하였으면, 이러한 사실을 알리는 ready 신호를 프레임 i에 전달하고, 프레임 i는 이 신호를 전달받은 후에 갱신된 루프 변수를 프레임 j에 전달한다.

이와 같은 순차 루프의 병렬 실행 방식을 고려한 최적 펼침도[14]는 (2)의 식과 같다.

$$K = \lfloor B/(\upsilon + t_\upsilon) \rfloor + \beta \qquad (2)$$

여기서 B는 한 반복의 계산시간(loop body 계산시간 포함)이고, υ는 반복 실행 초기부터 루프 제어변수 계산시까지의 소요 시간이고, t_υ는 값을 프레임간에 전달하는데 걸리는 시간이고, β는 다음번째 프레임으로부터 오는 ready 신호의 대기 시간이다. 루프 펼침도는 루프 본체 계산시간 B에 비례하고, 루프제어변수의 계산시간과 값을 프레임에 전달하는 시간에 반비례하는 것을 알 수 있다.

3.2 병렬 루프 펼침

병렬 루프는 반복간에 데이타나 제어의 종속이 존재하지 않고, 따라서 루프의 병렬 수행시에 반복간에 통신이 요구되지 않는다. 따라서 병렬 수행 환경을 위해서 할당된 프레임들간에 순차 루프처럼 체인을 형성할 필요가 없고, 이러한 사실은 loop-setup과 loop-exec이 중첩되어 수행 가능하다는 사실을 알려준다. 즉, loop-setup이 첫 번째 프레임을 설정하는 즉시, loop-setup이 다른 프레임들을 설정하는 동안에 loop-exec가 그 프레임 상에서 해당 반복을 수행할 수 있다는 것이다.

이와 같은 병렬 루프의 병렬 실행 방식을 고려한 최적 펼침도[14]는 (3)의 식과 같다.

$$K = \left\lceil \sqrt{\frac{NB}{dO_s}} \right\rceil \qquad (3)$$

여기서 N은 반복의 총 개수이고, d는 loop-setup과

loop-exec의 중첩 정도를 나타내고((0,1]의 범위를 갖는다), O_s는 loop-setup에 따른 부담이다. 루프 펼침도는 루프 계산시간 NB에 비례하고, 루프 펼침에 따른 부담 O_s에 반비례하는 것을 알 수 있다. 또한, 루프 펼침에 따른 부담은 루프 실행과 루프 병렬 수행 환경 설정의 중첩 정도에 따라 감소되는 것을 알 수 있다.

4. 중첩루프 펼침

3장에서는 [14]에 제시된 비정형성 함수형 언어 루프가 다중스레드 구조상에서 루프 특성을 반영하여 효과적으로 병렬 수행되는 방식을 살펴보았고, 또한 각 루프 특성에 따라서 계산된 최적 루프 펼침도를 살펴보았다. 그러나 [14]에서 계산된 최적 루프 펼침도는 단일 루프만을 고려한 것이다. 여기서는 동일한 루프 병렬 수행 방식 하에서, 중첩루프의 펼침을 고려하며, [14]의 단일 루프의 펼침 방법을 적용하는 방안도 함께 고려한다. 루프의 펼침은 기억공간과 프로세서 등과 같은 시스템 자원에 제한해서 이루어져야 한다. 특히, 병렬성이 상당히 많이 포함될 수 있는 중첩루프의 경우에, 시스템의 자원 양에 비해서 너무 지나치게 루프가 펼쳐질 경우에는 문제가 초래될 수 있다. 가령, 프로세서의 수에 비해서 루프가 지나치게 너무 많이 펼쳐질 경우에, 펼침의 부담이 커져서 병렬 수행의 역효과가 초래될 수 있으며, 기억공간이 부족할 경우에는 루프 펼침에 따른 이름공간을 확보하지 못하여 병렬 수행이 제대로 이루어질 수 없게 된다. 여기서는 먼저, 시스템 자원 제한을 고려하지 않은 루프 펼침을 살펴보고, 다음에 시스템 자원을 고려한 루프 펼침을 살펴본다. 마지막으로, 이 두 가지 사항을 고려한 중첩루프 펼침 알고리즘을 제안한다. 논문에서는 중첩루프에서 각 수준에서 중첩되어 있는 루프가 그 내부에 단지 한 개의 루프만을 포함하는 완전 중첩루프만을 고려한다.

4.1 자원을 고려하지 않은 루프 펼침

시스템 자원을 고려하지 않는 루프 펼침은 자원이 무한하다고 가정하는 것과 동일하다. 논문에서는 (2), (3)의 식을 사용하여 최내곽(innermost) 루프부터 시작하여 최외곽(outermost) 루프의 순서로 각 루프의 최적 펼침도를 계산한다. 루프 펼침도 계산을 위해서, 중첩루프를 트리로 표현한다. 루트 노드는 최외곽 루프를 나타내고, 그 자식 노드는 최외곽 루프에 포함된 루프를 나타낸다. 이와 같이 생각하면, 잎 노드는 최내곽 루프의 본체를 나타낸다. 논문에서는 각 수준의 루프가 단지 한 개의 루프만을 포함할 수 있으므로, 트리의 각 노드는 단지 한 개의 자식만을 갖는다. 트리의 각 노드는 $(N, B, \alpha, \beta, \sigma, K)$의 값을 가질 수 있다. 여기서 N은 루프의 반복구문수를 나타내고, B는 루프의 본체 계산시간을 나타낸다. α는 순차 루프의 경우에만 해당되는데 반복 실행 초기부터 루프 제어변수의 값을 갱신하고, 이 값을 다음번째 프레임에 전달하는데 걸리는 시간을 나타내는 것으로, 식 (2)에서 $(\upsilon+t_b)$의 값을 나타낸다. β도 순차 루프의 경우에만 해당되며, 식 (2)에서 β의 값을 나타낸다. σ는 한 개의 루프 프레임을 설정하는 부담을 나타내고, 식 (3)에서 dO_s의 값을 나타내고, K는 루프의 펼침도를 나타낸다. 순차 루프의 경우, N, B, α, β를 사용하고, 식 (2)를 이용하여 K를 계산하고, 병렬 루프의 경우 N, B, σ를 사용하고, 식 (3)을 이용하여 K를 계산한다. α, β, σ는 컴파일러가 성능 매개변수를 사용하여 계산할 수 있는 값이며, B는 최내곽 루프의 본체인 경우 계산될 수 있다.

알고리즘 CLUD_wo_RSC는 최내곽 루프의 본체를 나타내는 트리의 잎 노드를 제외한 각 노드에 대해서 루프 펼침도를 계산한다. 여기서는 각 루프에 대해서 N, α, β, σ가 컴파일러에 의해서 계산되어 있고, 최내곽 루프의 본체 B도 계산되어 있다고 가정한다. 중첩루프를 표현하는 트리의 루트 노드를 매개변수로 하여 CLUD_wo_RSC를 호출함으로써, 각 루프에 대해서 B와 K를 계산한다. 이 알고리즘은 *node*가 표현하는 루프의 펼침도, K를 계산하고, 자신을 K의 루프 펼침도로 펼쳐졌을 때의 본체 계산시간을 계산하고, 이것을 반환한다. 중첩루프에 포함된 각 루프의 펼침도는 최내곽 루프부터 시작하여 최외곽루프의 순서로 계산된다. 즉, 주어진 중첩루프 L을 표현한 트리의 루트 노드가 *root*일 때, CLUD_wo_RSC(*root*)를 수행함으로써 L의 펼침이 시작된다.

알고리즘 CLUD_wo_RSC이 어떻게 동작하는지 간단히 살펴보자. 먼저, *node*가 자식을 가지면 루프를 나타내는데, 이 루프의 본체를 계산하기 위해서 CLUD_wo_RSC을 그 자식을 매개변수로 하여 재귀적으로 호출한다. 그렇지 않으면, *node*는 잎 노드이며 최내곽 루프의 본체를 나타낸다. 이 본체는 미리 계산되어 있으므로, 단지 그 계산 시간을 반환한다. *node*의 B가 결정되었으면, 루프의 종류에 따라서 루프 펼침도를 계산한다. 순차루프는 (2)의 식을 사용하여 계산되고, 병렬루프는 (3)의 식을 사용하여 계산된다. 마지막으로, 계산된 루프 펼침도로 *node*의 루프를 펼쳤을 때의 병렬 계산시간(이것이 *node*의 B가 된다)을 계산하고 반환한다. 루프를 k의 펼침도로 펼쳐졌을 때의 병렬 계산시간은 (1)의 식을 사용하여 계산된다.

알고리즘 1: CLUD_wo_RSC(*node*)
// *node*는 트리의 한 노드이다.
// *node*는 $(N, B, \alpha, \beta, \sigma, K)$로 표현된다.
{
  IF (*node*가 자식을 가지면) THEN
    $B \leftarrow$ CLUD_wo_RSC(CHILD(*node*));
  ELSE // *node*가 잎 노드인 경우
    RETURN(B);
  // *node*의 루프 펼침도를 계산한다.
  IF (*node*가 병렬 루프이면) THEN
    식 (3)을 이용하여 k를 계산한다.
  ELSE
    식 (2)를 이용하여 k를 계산한다.
  // 현재 루프의 병렬 수행시간을 계산한다.
  $C \leftarrow NB/K + \sigma K$;
  RETURN C;
}

정리 1: 알고리즘 CLUD_wo_RSC는 임의의 주어진 중첩루프에 대해서 최적 루프 펼침도를 계산한다. 이 알고리즘의 계산복잡도(complexity)는 $O(n)$이다. 여기서 n은 L의 최대 중첩 수준을 나타낸다.

증명) 임의의 중첩루프 $L = (L_1, L_2, ..., L_n)$을 생각해보자. $L_i = (N_i, B_i)$로 표현될 때$(1 \le i \le n)$, N_i와 B_i는 L_i로 표현되는 루프의 반복구문수와 본체 계산시간을 각각 나타낸다. 증명은 CLUD_wo_RSC가 각 L_i에 대해서 B_i가 최소가 되는 최적 루프 펼침도 k_i를 계산한다는 것을 보임으로써 이루어진다. 이것은 중첩 수준의 관점에서 수학적귀납법을 적용하여 보일 수 있다. 먼저, i=1 경우는 단일 루프 펼침에 해당된다. 단일 루프는 순차루프에 대해서 식 (2)에 의해서, 병렬루프에 대해서는 식 (3)에 의해서 최적 펼침도를 각각 계산할 수 있다. 이제, 최내곽 루프부터 중첩 수준 $i(1<i<n)$에 해당되는 루프 L_i까지 최적으로 펼쳐졌다고 가정한다. 즉, 루프 펼침도 $u_i, u_{(i+1)}, ..., u_n$이 최적으로 계산되었다고 가정한다. CLUD_wo_RSC는 u_i와 식 (1)을 사용하여 $L_{(i-1)}$의 본체 계산시간 $B_{(i-1)}$를 계산한다. 이렇게 계산된 $B_{(i-1)}$는 가정으로부터 최소 계산시간이다. 다음에, $B_{(i-1)}$를 사용하여 중첩 수준 $(i-1)$의 루프 $L_{(i-1)}$의 루프 펼침도 $u_{(i-1)}$를 루프의 종류에 따라서 식 (2)나 (3)를 사용하여 계산한다. B_i는 최소 계산시간이고, $u_{(i-1)}$이 $L_{(i-1)}$의 최적 펼침도이다. 따라서 CLUD_wo_RSC는 중첩 루프 L에 포함된 모든 루프 $L_i(1 \le i \le n)$에 대해서 최적 루프 펼침도를 계산한다. 또한, CLUD_wo_RSC는 L에 속한 루프를 단지 한번씩 방문하므로, n이 L의 최대 중첩 수준을 나타낼 때, 그 계산복잡도는 $O(n)$이다.

■

4.2 자원 제한 루프 펼침

이미 언급하였듯이, 시스템 자원을 고려하지 않고서 루프를 펼친다면 자원의 부족으로 인하여 루프 펼침 효과가 떨어지거나 실행에 오류가 발생할 수 있다. 여기서는 기억공간과 프로세서의 시스템 자원을 고려한다. 루프의 펼침에는 새로운 루프 프레임의 할당을 요구한다. 루프 프레임은 반복구문의 실행 환경으로 해당 반복구문에 대한 이름공간을 제공한다. 루프 펼침도가 k일 경우에, k개의 루프 프레임을 할당하고 설정하게 된다. 그러나 k개의 루프 프레임을 할당할 기억공간이 확보되어 있지 않은 경우에, 기억공간 부족으로 인하여 프로그램 실행은 중단될 것이다. 따라서 사용가능한 기억공간의 크기에 제한하여 루프를 펼치는 것이 매우 중요하다. 특히, Id와 같은 비정형성 함수의 경우 미세한 수준의 대규모 병렬성을 내포하게 되므로, 현재의 시스템 자원을 고려하여 병렬성을 이용하는 방안이 고려되어야 한다.

최대 중첩 수준이 n인 중첩루프 $L = (L_1, L_2, L_3, ..., L_n)$을 생각해보자. 여기서 L_1은 최외곽 루프를 나타내고, L_n은 최내곽 루프를 나타낸다. 중첩수준이 i인 L_i를 생각해보자. L_i의 루프 프레임 크기가 m_i이고, 루프 펼침도가 k_i일 경우, L_i를 펼치는데 소요되는 기억공간 크기는 m_ik_i이다. 따라서 L을 펼치기 위해서 소요되는 기억공간의 크기는 다음과 같이 계산된다:

$$\prod_{1}^{n} m_i k_i \qquad (4)$$

여기서 $m_i < m_{(i+1)}$이 성립한다. 왜냐하면, 루프 프레임에는 루프 상수가 저장되는데, $L_{(i+1)}$의 루프 상수는 L_i의 루프 상수를 포함하기 때문이다. 여기서 루프 상수(loop constant)는 루프 실행시에 값이 변경되지 않는 변수를 의미한다. 프로그램 실행시에 중첩루프 L을 펼칠 시점에 현재 사용가능한 시스템 기억공간이 (4)의 크기보다 클 경우에 L은 펼쳐질 수 있다고 판단된다.

두 번째 고려사항은 프로세서의 개수이다. 위의 중첩루프 L에서 i번째 중첩 수준 루프 L_i의 루프 펼침도가 k_i이면, 이 L_i를 실행하는데 필요한 프로세서의 개수는 k_i이다. 따라서 L을 펼쳐서 실행하는데 필요한 프로세서의 개수는 다음과 같이 계산된다:

$$\prod_{1}^{n} k_i \qquad (5)$$

프로그램 실행중, 중첩루프 $L = (L_1, L_2, ..., L_n)$을 펼칠 시점에서 사용 가능한 프로세서의 개수가 P이고, 사

용 가능한 기억공간의 크기가 M일 때, 논문에서는 다음 두 가지 조건을 충족하는 L의 루프 펼침도 조합을 계산한다:

$$\prod_{1}^{n} m_i k_i \le M \tag{6}$$

$$\prod_{1}^{n} k_i \le P \tag{7}$$

위에서 (6)의 조건은 엄격하게 지켜져야 하는 것은 자명하다(그렇지 않으면, L의 병렬 수행은 기억공간의 부족으로 인하여 중단될 것이다). 그러나 (7)의 조건은 시스템의 구조와 프로그램의 병렬성 포함 정도에 따라서 어느 정도 완화될 수 있고, 오히려 그러한 경우가 더 최적인 펼침이 될 수 있다. 왜냐하면, 프로그램은 다중 스레딩 방식으로 실행되고, 원격에 위치한 값을 참조하거나 동기화를 수행하는 명령어는 긴 지체시간이 소요되고, 이러한 명령어의 응답이 도달할 때까지 프로세서는 대기하면서 유휴할 수 있고, 이때 실행가능한 다른 스레드가 존재한다면, 유휴중인 프로세서는 유용한 작업을 수행할 수 있고, 이는 병렬실행의 효과를 높일 수 있기 때문이다. 루프 펼침도가 클수록 스레드 큐상에 실행가능한 스레드가 존재할 가능성이 크므로, 루프 펼침도 조합이 P를 넘어서더라도 병렬수행 효과가 클 수 있다. 그러나 루프 펼침도의 크기에 비례해서 루프 펼침에 따른 부담이 커지므로 주의해야 한다. 따라서 최적 루프 펼침도 조합은 시스템과 프로그램의 성격에 따라서 상당히 달라질 수 있고, 이를 정확히 계산하는 것은 거의 불가능할 것이다. 여기서는 안전한 병렬 수행을 보장하면서 가능한 최적인 병렬 수행을 기대할 수 있는, 자원 제한 루프 펼침 알고리즘을 다음에 제시하는 보조정리 1, 2를 이용하여 제안한다.

보조정리 1: 중첩루프에서 내곽 루프보다는 외곽 루프를 펼치는 것이 병렬 수행에 더 효과적이다.

증명) 임의의 중첩루프 $L = (L_1, L_2)$를 가정한다. 여기서 $L_1 = (N_1, B_1)$, $L_2 = (N_2, B_2)$이고, L_2는 L_1의 내곽루프이다. N_1과 B_1은 L_1의 루프 경계와 본체 계산시간을 각각 나타내고, N_2와 B_2는 L_2의 루프 경계와 본체 계산시간을 각각 나타낸다. 동일 p개의 프로세서를 사용하여 L_1을 펼칠 경우의 실행시간을 T_1, L_2를 펼칠 경우의 실행시간을 T_2라고 하면, (1)의 식을 이용하여 T_1, T_2를 다음과 같이 표현할 수 있다.

$T_1 = N_1B_1/p + p\sigma_1$
$= N_1(N_2B_2)/p + p\sigma_1 = N_1N_2B_2/p + p\sigma_1$
$T_2 = N_1B_1$
$= N_2(N_2B_2/p + p\sigma_2) = N_1N_2B_2/p + N_1p\sigma_2$

위의 식에서 $\sigma_1 < \sigma_2$이 성립한다. σ_1, σ_2는 L_1, L_2의 루프 펼침 부담을 각각 나타내는데, 이러한 부담은 주로 loop-setup에 의해서 이루어지는 루프 병렬 수행 환경 설정에 기인한다. 루프 프레임 설정은 프레임을 할당하고, 여기에 루프 상수를 전달하여 저장하는 것으로 이루어진다. 그런데, 내곽 루프의 루프상수는 외곽루프의 루프상수에 자신의 루프상수를 추가하게 되므로, 외곽루프의 루프상수보다 더 많다. 따라서 프레임의 크기도 더 크고, 프레임을 설정하는데 더 많은 시간이 소요된다. 따라서 $\sigma_1 < \sigma_2$가 성립된다. 그러므로, 다음 식이 성립한다:

$T_2 - T_1 = (N_1N_2B_2/p + N_1p\sigma_2) - (N_1N_2B_2/p + p\sigma_1)$
$= p(N_1\sigma_2 - \sigma_1) > 0$ ($\because N_1, \sigma_1, \sigma_2 > 1$, 그리고 $\sigma_2 > \sigma_1$)

그러므로, $T_2 > T_1$이 성립된다. 그러므로, 내곽 루프보다는 외곽 루프를 펼치는 것이 병렬 수행에 더 효과적이다.

■

보조정리 1은 중첩루프 $L = (L_1, L_2)$를 펼칠 시점에서 프로세서 개수 P가 주어져 있을 때, P개의 프로세서를 가능한 내곽 루프보다 외곽 루프에 할당하는 것이 병렬 수행효과가 높다는 것을 보여준다. 그렇다면, P개의 프로세서를 모두 L_1에 할당하는 것이 최적인가? 보조정리 1은 이 질문에 대해서는 답변해 주지 않는다. 즉, 보조정리 1은 P를 어떠한 비율로 분할하여 L_1, L_2에 할당하는 것이 효과적인지에 대해서는 설명하지 않는다. 다음에 제시된 보조정리 2는 이러한 질문에 대해서 답변한다.

보조정리 2: 중첩루프 $L = (L_1, L_2)$에서, $L_1 = (N_1, B_1, \sigma_1)$, $L_2 = (N_2, B_2, \sigma_2)$이고, L_1, L_2의 루프 펼침도를 각각 k_1, k_2이고, 프로세서의 개수가 P일 때, $k_1k_2 \le P$를 만족하면서, L_1, L_2에 최적으로 할당하는 방법은 다음과 같다. 여기서 N_1, B_1, σ_1은 L_1의 반복구문 수, 본체 계산시간, 루프 프레임 설정 시간을 각각 나타낸다:

$$k_1 = \left\lfloor \sqrt[3]{\frac{2N_1P\sigma_2}{\sigma_1}} \right\rfloor \tag{8}$$

$$k_2 = \left\lfloor \sqrt[3]{\frac{P^2\sigma_1}{2N_1\sigma_2}} \right\rfloor \tag{9}$$

증명) L_1을 x $(1 \le x \le P)$의 루프 펼침도로 펼쳐서 수행시킨 계산시간을 $T_{k_1}(L_1)$이라 하고, L_2를 $\lfloor P/x \rfloor$의 루프 펼침도로 펼쳐서 수행시킨 계산시간을 $T_{k_2}(L_2)$이라 하면, 다음 식이 성립한다:

$$T_{k_1}(L_1) = \frac{N_1B_1}{x} + x\sigma_1,$$

$$T_{k_2}(L_2)=\frac{N_2B_2}{\left\lfloor\frac{P}{x}\right\rfloor}+\left\lfloor\frac{P}{x}\right\rfloor\sigma_2$$

$B_1 = T_{k_2}(L_2)$이므로,

$$T_{k_1}(L_1)=\frac{N_1}{x}\left(\frac{N_2B_2}{\left\lfloor\frac{P}{x}\right\rfloor}+\left\lfloor\frac{P}{x}\right\rfloor\sigma_2\right)+x\sigma_1$$

$$\approx \frac{N_1}{x}\left(\frac{N_2B_2x}{P}+\frac{P}{x}\sigma_2\right)+x\sigma_1$$

$$=\frac{N_1N_2B_2}{P}+\frac{N_1P\sigma_2}{x^2}+x\sigma_1$$

$$\overset{let}{=} f(x)$$

$f(x)$를 x에 대해서 미분한 결과를 $f(x)'$이라 하면,

$$f(x)'=\frac{-2N_1P\sigma_2}{x^3}+\sigma_1 \overset{let}{=} 0$$

$x=\sqrt[3]{\frac{2N_1P\sigma_2}{\sigma_1}}$ 일 때, $f(x)$는 최소값을 갖는다.

따라서, $k_1=\lfloor x\rfloor=\left\lfloor\sqrt[3]{\frac{2N_1P\sigma_2}{\sigma_1}}\right\rfloor$ 일 때,

$T_{k_1}(L_1)$은 최소값을 갖는다.

마찬가지로, $k_2=\left\lfloor\frac{P}{x}\right\rfloor=\left\lfloor\sqrt[3]{\frac{P^2\sigma_1}{2N_1\sigma_2}}\right\rfloor$ 일 때,

$T_{k_2}(L_2)$는 최소값을 갖는다.

또한, $k_1k_2 = \lfloor x\rfloor\lfloor P/x\rfloor \le x(P/x) = P$이므로,

$k_1k_2 \le P$를 만족한다.

■

논문에서는 보조정리 1과 2를 이용하여 주어진 시스템 자원에 제한해서 임의의 주어진 중첩루프를 펼치는 알고리즘 CLUD_NestedLoop를 제안한다. CLUD_NestedLoop가 주어진 중첩루프 $L = (L_1, L_2, ..., L_n)$을 펼친다고 가정한다. 먼저, 알고리즘 1에서 제안한 CLUD_wo_RSC를 이용하여 자원을 고려하지 않고 루프를 최적으로 펼칠 수 있는 루프 펼침도 조합을 계산하고, 이러한 펼침에 필요한 자원이 존재하면, L을 최적으로 펼친다. 그렇지 않으면, 알고리즘 CLUD_w_RSC를 이용하여 주어진 시스템 자원에 제한하여 L을 다시 펼친다. 이때 CLUD_w_RSC는 시스템의 자원, 프로세서와 기억공간을 고려하면서 L을 펼친다. 따라서 자원을 고려하지 않는 CLUD_wo_RSC와는 달리, 시스템 자원 P와 M의 매개변수를 추가로 전달받는다. 여기서 P는 사용 가능한 프로세서의 수를 나타내며, M은 사용 가능한 시스템의 기억공간 크기를 나타낸다. 또한, M_c를 매개변수로 전달받는데, 이것은 이 알고리즘이 주어진 중첩루프에 대해서 이전 중첩수준까지의 루프를 펼치는데 필요한 기억공간의 크기를 나타낸다.

CLUD_w_RSC는 보조정리 1의 사실을 반영하여 최외곽 루프부터 최내곽루프의 순서로 루프를 펼치기 시작하며, 시스템 자원이 충분하지 않으면 내곽 루프의 펼침을 중단한다. 이 알고리즘이 어떻게 중첩루프를 펼치는지를 살펴보자. CLUD_w_RSC는 먼저, 보조정리 2를 이용하여 P를 현재의 루프와 다음번째 내곽 루프에 대해서 최적 분할한다. 현재 루프를 L_1, 다음번째 내곽 루프를 L_2라 하고, 보조정리 2에 의해서 L_1, L_2에 할당된 프로세서 수를 k_1, k_2라 하면, $k_1k_2 \le P$가 만족된다. $k_2 < 2$이면 L_1만 펼치게 되며, L_1 중첩수준까지 펼치는데 필요한 기억공간을 계산하고, 이것을 사용 가능한 기억공간과 비교한다. L_1을 k_1의 펼침도로 펼치는데 기억공간이 부족하면, 사용가능한 기억공간 범위내에서 최대로 펼칠 수 있는 k_1을 다시 계산한다. 보조정리 2에서 계산된 k_2가 2이상이면, L_1, L_2가 모두 펼쳐진다. 계산된 펼침도로 L_2 중첩 수준까지 루프를 펼치는데 필요한 기억공간이 존재하면, CLUD_w_RSC를 재귀적으로 호출한다. 이때 전달되는 매개변수는 L_2, k_2, M, M_c이다. 재귀적으로 호출된 CLUD_w_RSC는 L_2에 속한 루프 L_3가 존재할 경우, 마찬가지로 보조정리 2를 이용하여 k_2를 L_2, L_3에 대해서 분할할 것이다. 만약에, L_2의 중첩수준까지 펼칠 기억공간이 충분하지 않으면, 보조정리 1을 적용하여 내곽 루프, 즉 L_2의 펼침도부터 감소시키면서, k_1, k_2를 순서대로 재조정한다.

알고리즘 2: CLUD_NestedLoop(*root, P, M*)

```
// root는 중첩루프 L = (L1, L2, ..., Ln)을 표현한다.
// P는 사용 가능한 프로세서의 개수
// M은 시스템의 사용 가능한 기억공간의 크기
{
   // 먼저, 자원을 고려하지 않고 루프를 펼친다.
   CLUD_wo_RSC(root);

   // 최적 펼침을 위해서 필요한 자원, ML, PL을 계
      산한다.
   ML = ∏_1^n mi ki;
```

// m_i는 L_i의 루프 프레임 설정 시간이고,
// k_i는 CLUD_wo_RSC에서 계산된 L_i의 루프 펼침 도이다.

$$P_L = \prod_1^n k_i;$$

IF ($M_L > M$ or $P_L > P$) THEN
// 시스템 자원이 충분하지 않으면
// 자원에 제한해서 루프를 다시 펼친다.
CLUD_w_RSC(*root, P, M,* 1);
END IF
}

알고리즘 3: CLUD_w_RSC(*node, P, M, M_c*)
// *node*는 트리의 한 노드이고,
// *node*는 (*m,u,k*)의 자료구조로 표현된다. *m*은 루프 프레임의 크기이고,
// *u*는 CLUD_wo_RSC에서 할당된 최적 루프 펼침 도이고,
// *k*는 이 알고리즘에서 계산될 실제 루프 펼침도
// *P*는 사용 가능한 프로세서의 개수
// *M*은 시스템의 사용 가능한 기억공간의 크기
// M_c는 알고리즘에서 할당한 기억공간의 크기
{
IF (*node*의 자식이 루프이면) THEN
// L_1은 *node*가 나타내는 루프이고, k_1은 L_1의 루프 펼침도
// L_2는 *node*의 자식이 나타내는 루프, k_2는 L_2의 루프 펼침도
// *P*를 k_1, k_2로 분할, $k_1*k_2 \le P$를 만족
식 (8), (9)를 사용하여 *P*를 L_1, L_2에 k_1, k_2로 분할하여 할당
$M_1 \leftarrow L_1(m)*k_1$ // L_1을 k_1의 펼침도로 펼치는데 필요한 기억공간
$M_c \leftarrow M_1*M_c$ // L_1 중첩 수준까지 펼치는데 필요한 기억공간
IF ($k_2 < 2$ and $M_c > M$) THEN
// L_1만 펼침
$M_c \le M$을 만족하는 최대값 k_1을 결정
$k_1 \leftarrow min(k_1, L_1(u))$
ELSE IF ($k_2 \ge 2$) THEN
$M_2 \leftarrow L_2(m)*k_2$ // L_2를 k_2의 펼침도로 펼치는데 필요한 기억공간
$M_c \leftarrow M_c*M_2$; // L_2 중첩 수준까지 펼치는데 필요한 기억공간
IF ($M_c > M$) THEN
// 기억공간이 부족하면
$M_c \le M$을 만족하는 최대값 k_1, k_2를 다시 계산
ELSE
CLUD_w_RSC(CHILD(*node*), k_2, *M*, M_c)
END IF
END IF
END IF
}

정리 2: 임의의 중첩루프 $L = (L_1, L_2, ..., L_n)$에 대해서 알고리즘 CLUD_NestedLoop이 수행하는 루프 펼침은 식 (6)과 (7)의 두 가지 조건을 충족한다. 또한, 이 알고리즘의 계산복잡도는 O(*n*)이다. 여기서 *n*은 *L*의 최대 중첩 수준을 나타낸다.

증명) 임의의 중첩루프 $L = (L_1, L_2, ..., L_n)$을 생각하자. CLUD_NestedLoop의 수행 결과로, $L_i(1 \le i \le n)$가 (m_i, k_i)의 값을 갖는다고 하자. 여기서 m_i는 L_i의 루프 프레임의 크기이고, k_i는 알고리즘의 수행 결과로 L_i에 할당된 프로세서의 개수이다. 증명은 다음과 같이 CLUD_NestedLoop이 최적 할당을 수행한 경우와 그렇지 않은 경우로 구분하여 이루어진다:

i) CLUD_NestedLoop이 최적 할당을 수행하는 경우를 살펴보자. 최적 할당은 CLUD_wo_RSC를 호출하여 자원을 고려하지 않고 *L*을 펼치는 경우이다. 그 결과로, 다음 두 조건을 만족하면,

$$\prod_1^n m_i k_i \le M, \quad \prod_1^n k_i \le P$$

CLUD_NestedLoop은 *L*에 대해서 (6), (7)의 두 가지 조건을 만족하면서 *P*의 최적 할당을 수행한다.

ii) 최적 할당이 실패한 경우, CLUD_NestedLoop은 CLUD_w_RSC를 호출하여 주어진 자원에 제한하여 루프를 펼친다. 이러한 루프 펼침이 (6), (7)의 조건을 만족한다는 것을 중첩 수준의 관점에서 수학적 귀납법을 사용하여 증명하고자 한다. 먼저, *l*의 중첩수준의 루프까지 할당된 자원이 다음을 만족한다고 가정한다.

$$\prod_1^l m_i k_i \le M, \quad \prod_1^l k_i \le P$$

이때, CLUD_w_RSC를 재귀적으로 호출하여 (*l*+1)의 중첩수준에 속한 루프를 펼치고자 한다. 전달된 매개변

수는 $L_{(l+1)}$, k_l, M_c이다. 여기서 $M_c = \prod_1^l m_i k_i$

CLUD_w_RSC는 보조정리 2를 이용하여 k_l을 $k_{l'}$, $k_{(l+1)}$로 분할하고, L_l에는 $k_{l'}$을, $L_{(l+1)}$에는 $k_{(l+1)}$의 프로세서 수를 할당한다. 여기서 $k_l k_{(l+1)} \le k_l$이 만족된다. 따라서 다음이 만족된다.

$$\prod_1^{l+1} k_i = k_1 \cdot k_2 \cdot \ldots \cdot k_{l'} \cdot k_{(l+1)} \le k_1 \cdot k_2 \cdot \ldots \cdot k_l = \prod_1^{l} k_i \le P \quad ①$$

두 번째로, CLUD_w_RSC는 다음을 만족하는 경우에만, $L_{(l+1)}$의 펼침을 허용한다.

$$\prod_1^{l+1} m_i k_i \le M \quad ②$$

①, ②의 두 사실로부터 CLUD_w_RSC가 수행한 L에 대한 루프 펼침은 (6), (7)의 두 가지 조건을 만족한다. 그러므로, CLUD_NestedLoop이 수행한 L에 대한 루프 펼침은 (6), (7)의 두 가지 조건을 만족한다.

CLUD_NestedLoop에서 CLUD_wo_RSC의 계산복잡도는 정리 1로부터 $O(n)$이다. 여기서 n은 L의 최대 중첩 수준을 나타낸다. CLUD_w_RSC은 L에 속한 각 루프를 최외곽 루프부터 시작하여 많아야 단지 한번만 방문하게 되므로, 그 계산복잡도는 $O(n)$이다. 따라서 CLUD_NestedLoop의 계산복잡도는 $O(n)$이다.

■

5. 결 론

본 논문에서는 다중스레드 구조에서 비정형성 함수형 언어의 중첩루프에 포함한 병렬성을 효과적으로 이용할 수 있는 알고리즘 CLUD_NestedLoop을 제안하였다. 이 알고리즘은 먼저, 자원을 고려하지 않고서 최적 중첩루프 펼침도 조합을 계산하고, 현재의 시스템 자원이 이 중첩루프 펼침도 조합을 지원할 수 있는지의 여부를 판단하고, 그렇지 않으면, 시스템 자원에 제한해서 그 중첩루프를 다시 펼친다. 이 알고리즘은 안전하면서도 효과적으로 중첩루프를 펼칠 수 있는 특징을 지니며, 또한 중첩루프를 펼치는 시점에 시스템의 자원을 고려한다는 점에서 동적으로 사용될 수 있다. 향후 연구과제로는, 제안된 알고리즘을 실제 프로그램에 적용하여 그 효과를 분석하는 것이다. 또한, 제안된 알고리즘은 중첩루프에서 각 중첩 수준의 루프는 단지 한 개의 루프만을 포함할 수 있다는 제한을 두고 있는데, 일반적인 중첩루프에 대해서 제안된 알고리즘을 확장하고자 한다. 마지막으로 제한된 시스템 자원하에서도 최적인 중첩루프 펼침에 대해서 연구하고자 한다.

참 고 문 헌

[1] Arvind and R.A. Ianucci, "Two fundamental issues in multiprocessing," Technical Report, CSG Memo 226-5, Lab. for Computer Science, MIT, 1986.

[2] R.A. Ianucci, Parallel Machines: Parallel Machine Languages, Kluwer Academic Publishers, 1990.

[3] R.S. Nikhil, G.M. Papadopoulous and Arvind, "*T: A multithreaded massively parallel architecture," In Proc. of the 19th Annual Internatinal Symposium on Computer Architecture, pp. 159-167, 1992.

[4] D.E. Culler, S.C. Goldstein, K.E. Schauser and T. Eicken, "TAM- A compiler controlled threaded abstract machine," Journal of Parallel and Distributed Computing, Vol. 18, No.3, pp.347-370, 1993.

[5] Y. Kodama, H. Sakai, et al., "The EM-X Parallel Computer: Architecture and Basic Performance", In Proc of the 22nd Annual ISCA, pp. 14-23, 1995.

[6] S. Ha, J. Kim, et al., "A Massively Parallel Multithreaded Architecture: DAVRID," in Proc. of IEEE International Conference on Computer Design: VLSI in Computers and Processors, pp. 70-74, 1994.

[7] B.S. Ang, D. Chiou, et al., "Message passing support on StarT-Voyager, "Technical Report, CSG Memo 387, Lab. for Computer Science, MIT, 1996.

[8] R.S. Nikhil, "Id reference manual, Version 90.1," Technical Report, CSG Memo 284-2, Lab. for Compuer Science, MIT, 1990.

[9] Shail Aditya, Arvind and Jan-Willem Maessen, "Semantics of pH: A parallel dialect of Haskell, CSG Memo 377-1, 1995.

[10] K.R. Traub, et al., "Global analysis for partitioning non-strict programs into sequential threads," In Proc. of ACM Conf. on LISP and Functional Programming, pp. 324-334, 1992.

[11] K.E. Schauser, D.E. Culler and T. Eicken, "Compiler-controlled multithreading for lenient parallel languages, " In Proc. of the 5th Functional Programming and Computer architecture, pp.50-72, 1991.

[12] J.E. Hoch, et al., "Compile-time Partitioning of a Non-strict Language into Sequential Threads", In Proc. of the 3rd IEEE Sympo. on Parallel and Distributed Processing, pp. 180-411, 1989.

[13] E. Roh, S. Ha, et al., "Compilation of a functional language for the multithreaded architecture: DAVRID," In Proc. of International Conference on

Parallel Processing, Vol. 2, pp. 239-242, 1994.
[14] Sangho Ha and Heunghwan Kim, "KU-Loop Scheme: An Efficient Loop Unfolding Scheme for Multithreaded Computation", Journal of Information Science and Engineering, Vol. 14, No. 1, March 1998.
[15] C. D. Polychronopoulos, Parallel Programming and Compilers, Reading, Kluwer Academic Publishers, 1988.
[16] M. J. Wolfe, "Optimizing Supercompilers for Supercomputers," Ph.D. Thesis, Univ. of Illinois at Urbana-Champaign, DCS Report No. UIUCCDCS-R-82-1105, 1982.

하 상 호

1988년 서울대학교 계산통계학과(학사). 1991년 서울대학교 계산통계학과(석사). 1995년 서울대학교 전산학과(박사). 1995년 ~ 1996년 한국전자통신연구원 박사후 연구원. 1996년 ~ 1997년 MIT LCS 박사후 연구원. 1997년 ~ 현재 순천향대학교 정보기술공학부 조교수. 관심분야는 프로그래밍언어, 병렬 및 분산처리, XML.