□기술해설□

병렬처리를 위한 동기화 기법

순천향대학교 박두순*
고려대학교 이광형***
고려대학교 황종선*
건양대학교 김병수**

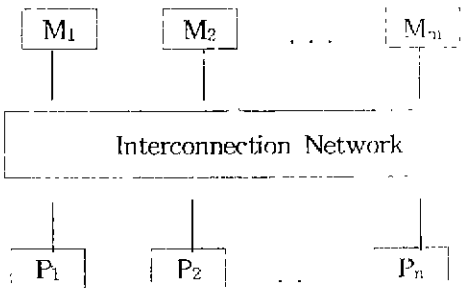
목 차

1. 서 론

폰노이만 방식의 컴퓨터 시스템은 주어진 연산을 순차적으로 수행하기 때문에 실행 속도를 빠르게 하는데는 많은 제약을 갖게된다. 이에 따라 실행 결과를 빠르게 얻고자 하는 고속 연산에 대한 욕구가 날로 증가함에 따라 순차 처리기로부터 고도의 병렬 실행이 가능한 벡터 및 병렬 처리 시스템의 필요성이 대두되었으며, IBM 3090/VF[7]와 같은 고도의 파이프라인(pipeline)으로 연결된 벡터(vector) 컴퓨터와 Cray X-MP[7], Cray-2[19], Ultracomputer [14], Cedar[12][18], Alliant FX/8[3] 시스템처럼 여러개의 벡터 처리기들이 한 시스템내에서 상호 연결되어 프로그램을 병렬처리 할 수 있는 형태의 시스템들이 등장하였다.

멀티프로세서의 기본적인 블럭 다이어그램은 그림 1과 같다.

그림 1에서, 각 프로세서는 interconnection network를 통해서 각 메모리 모듈을 액세스할 수 있다.



P_j : 프로세서 j

M_i : 메모리 모듈 i

그림 1 간단한 멀티프로세서 시스템 구조

그러나, 단순히 더 많은 프로세서를 시스템에 부착하는 것만으로는 시스템의 성능 향상에 커다란 영향을 미치지 못한다. 따라서 대규모 병렬처리 시스템상에서 처리 속도의 극대화를 꾀

*종신회원
**정 회 원
***학생회원

하기 위해서는 하드웨어 뿐만아니라 병렬성 추출 알고리즘과 같은 소프트웨어적인 요소들이 함께 고려되어야 한다. 이러한 연구의 일환으로 기존의 프로그래밍 환경과는 다른 병렬성을 표현할 수 있는 새로운 프로그래밍 환경에 대한 연구[24], 병렬 언어에 대한 연구 즉, Fortran이나 C와 같은 언어를 병렬 프로그램용으로 확장하는 것이나[5], 병렬 프로그램을 위하여 새로운 언어의 설계[9][10][15]에 대한 연구가 진행되고 있다.

그러나 직접 병렬 언어로 프로그램을 작성하는 것이나 이미 제작된 순차 프로그램을 새로운 병렬 언어로 수정하는 작업은 주어진 병렬처리 시스템의 구조를 이해하고 병렬 처리에 관한 전문지식을 새로이 익혀야 하는 등 프로그래머에게 상당한 부담이 되기 때문에 순차 프로그램을 자동적으로 병렬 처리가 가능한 형태로 변환해 주는 재구조화 기법이 많이 연구되고 있다.

재구조화 기법을 이용하여 프로그램의 병렬성(parallelism)을 추출하기 위해서는 먼저 프로그램의 데이타 종속성(data dependence)을 분석해야 한다[23].

종속성 분석은 주로 루프(loop)의 구조를 대상으로 이루어지고 있다. 왜냐 하면 루프는 잘 정의된 구조(well-defined structure)로써 데이타 종속성에 대한 분석이 용이하며, 비교적 작은 단위의 모듈이기 때문에 잠재적인 병렬성 획득을 위한 연산이 비교적 간단하기 때문이다. 또한 응용 프로그램에서 루프는 대부분의 수행시간(약 80%)을 차지하기 때문에 병렬성 추출의 핵심 부분이라 할 수 있다[1][17].

루프의 형태는 모든 반복(iteration)들간에 종속성이 없는 경우(do_all)와 서로 다른 반복간에 종속성이 발생하는 경우(do_across)가 존재한다(그림 2 참조). Do_all 형태의 루프에서는 서로 다른 반복간에 종속관계가 존재하지 않기 때문에 한 반복내의 모든 데이타를 프로세서간의 어떠한 상호 작용(interaction)없이 병렬처리 할 수 있으나, do_across 루프에서는 한 반복에서 가져온 데이타가 다른 반복에서 수정되거나, 한 반복에서 생성된 결과가 나중에 다른 반복에서 사용되는 종속 관계가 존재하기 때문에 완전한 병렬 처리를 수행할 수 없다. 즉, 부분적으로 프로세서간의 정보 교환이 필요하게 된다. 이러한 프로세서간의 상호 정보 교환을 동기화 기법이라 하며 프로그램의 올바른 수행을 위해서 반드시 유지되어야 한다.

그림 2의 (a)에서, 문장 S1과 S2상의 배열 변수 A는 동일한 반복내에서만 종속 관계가 성립하기 때문에 do_all 루프에 해당하고, (b)에서는 S1에서 정의된 변수 A는 이후의 반복에서 다시 문장 S2에 의해 재사용되기 때문에 do_across 루프에 해당한다. 따라서 do_across 루프를 병렬 처리할 때에는 프로세서간의 동기화가 유지되어야 한다.

```
do i = 1, N
S1: A[i] = B[i] × SQRT(X[i])
S2: D[i] = A[i] + C[i]
end do
```

(a) do_all 루프의 예

```
do i = 1, N
S1: A[i] = B[i] × 2
S2: D[i] = A[i-2] + C[i]
end do
```

(b) do_across 루프의 예

그림 2 do_all, do_across 루프의 예

본 연구에서는 공유 메모리를 갖는 멀티프로세서 시스템상에서 루프의 병렬 처리 시 데이타간의 종속성을 유지하기 위한 동기화 기법의 종류 및 특성을 설명한다.

일반적으로 동기화 기법은 동기화 수행 방법에 따라 임의(random) 동기화, 장벽(barrier) 동기화, 파이프라인(pipeline) 동기화 그리고 임계 영역(critical section) 동기화 기법으로 나눌 수 있다. 또한 처리 대상이 되는 루프내의 데이타간의 종속 형태에 따라 불변 종속 거리(constant dependence distance) 동기화와 가변 종속 거리(variable dependence distance) 동기화 기법으로 나눌 수 있다.

본 논문의 구성은 다음과 같다. 2장에서는 동기화하는 데 꼭 필요한 데이타 종속성을 살펴보고, 3장에서는 동기화 기법들에 대해서 설명한다. 마지막으로 4장에서는 결론을 맺는다.

2. 데이타 종속성

프로그램의 종속성은 자료(data) 종속성과 제어(control) 종속성으로 구성된다. 제어 종속성은 조건 분기에 의해서 발생되는 종속성으로 데이타 종속성과 유사한 방법으로 처리할 수 있기 때문에[20] 본 연구에서는 자료 종속성에 대해서만 설명한다.

데이타 종속성은 흐름 종속(flow dependence : read-after-write), 반 종속(anti dependence : write-after-read) 그리고 출력 종속(output dependence : write-after-write)으로 나눌 수 있다[4].

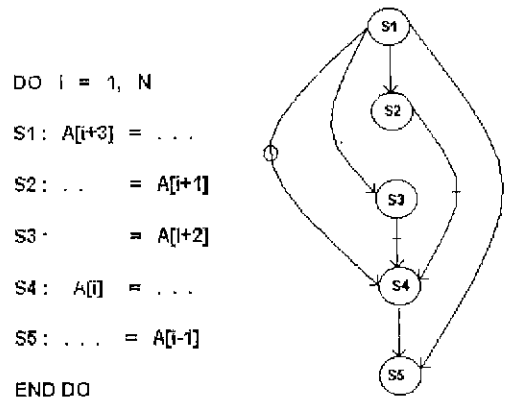
두 문장 S_i와 S_j 사이에서, S_i에서 변수 X가 선언(define)되고 S_j에서 X가 사용(use)되며 S_j가 S_i 이전에 수행되면 흐름 종속성($Si\,\delta\,Sj$)이 존재하고 두 문장 S_i와 S_j에서, 같은 변수가 선언되고, S_i가 S_j 이전에 수행되면 출력 종속성($Si\,\delta^o\,Sj$)이 존재하며 문장 S_i에서 X를 사용하고 S_j에서 X가 정의되며 S_i가 S_j 이전에 수행되면 반 종속성($Si\,\delta^a\,Sj$)이 존재한다.

이 중에서 흐름 종속은 참 종속(true dependence)이라 하고, 다른 두가지 종속은 인위 종속(artificial dependence)이라 한다. 그 이유는 출력 종속과 반 종속은 프로그램을 변환함으로써 제거할 수 있으나, 흐름 종속은 제거할 수 없기 때문이다[16].

이러한 데이타 종속성은 병렬성을 추출하고 프로그램 재구조화를 수행할 때 반드시 고려해야 할 중요한 개념이다. 즉, 데이타 액세스를 위한 순서를 나타내며 올바른 수행 결과를 얻기 위해 반드시 유지되어야 한다.

데이타 종속성과 제어 종속성을 함께 표현할 수 있는 그래프 중간 표현 형태로 종속 그래프(dependence graph)가 사용된다. 이 종속 그래프는 G=(V, E) 형태의 방향성 그래프(directed graph)인데, 여기서 V(Vertex 또는 Node)는 그래프상의 노드로서 한 문장이나 기본 블럭(basic block) 단위가 될 수 있다. 그리고 E(Edge)는 두 노드사이의 데이타 종속성이나 제어 종속성을 표현하는 방향성 에지이다. 에지의 머리에 해당하는 데이타를 source 데이타 그리고 에지의 꼬리에 해당하는 데이타를 sink 데이타라 한다.

그림 3의 종속 그래프에서는 4개의 흐름 종속($S1\,\delta\,S2$, $S1\,\delta\,S3$, $S1\,\delta\,S5$, $S4\,\delta\,S5$), 2개의 반 종속($S2\,\delta^a\,S4$, $S3\,\delta^a\,S4$), 그리고 하나의 출력 종속($S1\,\delta^o\,S4$)이 존재한다. 여기서, →는 흐름 종속(flow dependence), −|→는 반 종속(anti dependence) 그리고 −0→는 출력 종속(output dependence)를 의미한다.



(a) 예제 루프 (b) 종속성 그래프

그림 3 루프의 종속성 그래프

데이타 종속 거리(data dependence distance)는 최적화된 병렬 루프를 변환하는데 중요한 정보로 사용되며, 첨자식에 따라 불변 종속 거리와 가변 종속 거리로 나눌 수 있다. 만약 하나의 루프 인덱스 변수로만 구성된 첨자식이 각각 ai + b, ci + d(여기서 a, b, c, d는 정수, i는 루프 변수)라 하면, 다음과 같이 종속 거리(△)를 구할 수 있다. 위의 첨자식에서 a = c이면, 종속 거리는 |(ai ± b) − (ci ± d)|이고 인스턴스 사이의 값이 일정한 불변 종속 거리가 존재하고 a ≠ c이면, 인스턴스 사이에 가변 종속 거리가 존재한다. 불변 종속 거리 종속은 하나의 종속 거리만 존재하게 되고, 가변 종속 거리 종속은 여러개의 종속 거리가 존재하게 된다.

3. 동기화 기법

프로그램내에는 복잡한 데이타 종속을 갖는 반복(recurrence) 루프나 복수 지정문 등을 포

함하는 루프들과 같이 재구조화 기법들을 적용할 수 없는 루프가 상당히 존재한다. 비록 적용할 수 있다 하더라도 문장 재구성 기법 적용시 대개의 경우 본래의 순차 프로그램보다 프로그램 코드수가 늘어나며, 조건 조사, 최소값 적용, 최대 공약수 및 최소 공배수 등의 계산에 소요되는 오버헤드가 발생하게 된다.

동기화 기법은 이러한 결점들을 보완하는 유력한 방법으로 지난 수년간 폰 노이만 방식의 SISD 및 SIMD 구조에서 연구되어온 분야이며 최근에는 MIMD 시스템에서 병렬처리를 위한 기법으로 많이 연구되고 있다.

이러한 기법으로 임의 동기화 기법, 장벽 동기화 기법, 파이프라인 동기화 기법 그리고 임계 영역을 이용한 동기화 기법 등이 개발되어 사용되고 있다.

본 장에서는 루프의 병렬처리를 위하여 제안된 기존의 동기화 기법의 종류 및 특성을 파악한다. 특히 일반적으로 가장 많은 연구가 되고 있는 임의 동기화 기법에 주안점을 둔다.

3.1 임의 동기화 기법(Random Synchronization Scheme)

임의 동기화 기법[29]은 컴파일러가 루프의 반복들간에 교차된 데이타 종속이 존재하면 각 종속에 대하여 동기화 명령어들을 삽입하는 기법으로, 공유 메모리를 이용하는 임의 동기화 기법은 주로 세마포어(semaphore)에 기초한다. 즉, 루프내의 문장에서 source 문장 다음에 해당 문장이 완료되었음을 나타내는 동기화 명령을 삽입하고, sink 문장 바로 전에 해당 문장(source)이 완료되기를 기다리게 하는 동기화 명령을 삽입함으로써 병렬처리 시의 올바른 수행 순서를 유지한다.

임의 동기화 기법에는 데이타 종속 거리의 형태에 따라 불변 종속 거리 동기화 기법과 가변 종속 거리 동기화 기법으로 나눌 수 있다.

3.1.1 불변 종속 거리 동기화 기법

불변 종속 거리 기법은 병렬성을 추출하는 크기에 따라 데이타 지향(data-oriented) 기법, 문장 지향(statement-oriented) 기법 그리고 프로세스 지향(process-oriented) 기법 등으로 나누어 진다[27].

(1)데이타 지향 기법

데이타 지향 기법은 데이타의 올바른 접근 순서(access order)를 유지하기 위해서 데이타 종속을 이루는 source 데이타 각각의 모든 인스턴스에 동기화 변수, AC(Access Counter)를 할당한다.

종속 관계를 갖는 인스턴스(source 및 sink 인스턴스)를 수행하기 위해서는 먼저 source 인스턴스를 수행한 후, 수행 완료를 표시하기 위해서 해당 AC의 값을 수정한다. 또한 sink 인스턴스를 수행하기 위해서, 우선 해당 source 인스턴스가 완료 되었는가를 확인한 후(AC의 값 확인) 수행 여부를 결정한다.

그림 4는 그림 3의 (a)에서 제시한 루프에 대한 데이타 지향 동기화 수행 방법을 나타낸 것이다.

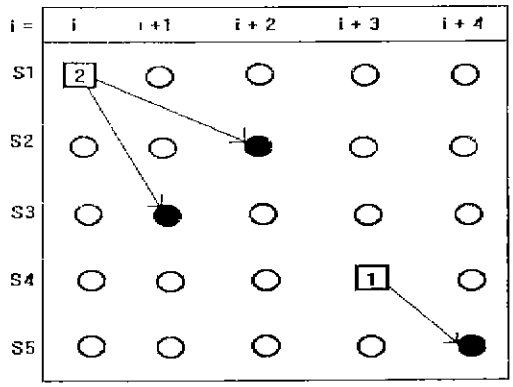


그림 4 데이타 지향 동기화 기법의 수행 방법

위의 그림에서 문장 S2와 S3의 읽기 연산을 허용하기 위해 S1의 수행이 완료된 후 해당 동기화 변수를 full 시키고, S2와 S3는 해당 동기화 변수가 full이 될 때까지 대기한 후 수행한다.

데이타 지향 기법으로는 Denelcor HEP의 Full/Empty tag 기법[20][26]등이 있으며 Extended Full Empty 동기화 명령어에 대한 간단한 설명은 다음과 같다.

Full/Empty 동기화 기법(FES)은 초기에 Denelcor HEP[26]에서 구현되었다. 이 기법에서 종속 관계를 갖는 데이타의 각 인스턴스는 동기화 변수로써 하나의 one-bit key와 연관을 갖는다. 이 키는 해당 데이타가 full 또는 empty인지를 나타낸다. 즉, 임의의 한 인스턴스가 동기화 읽기-연산을 수행할 때 해당 키는 empty가 되고 쓰기-연산을 수행할 때 full이 된다. 따라서 키의 상태를 확인함으로써 읽기 또는 쓰기-연산을 수행할 때 해당 작업이 유효한가를 결정할 수 있다. 그러나 FES 기법의 단점은 각 동기화 변수는 두가지 상태만을 표현할 수 있다는 점이다. 만약 루프내에서 읽기 또는 쓰기-연산이 여러번 발생하는 경우는 동기화를 수행할 수 없게 된다. 즉, 그림 5의 (a) 루프에서는 쓰기, 읽기 그리고 다시 쓰기-연산을 수행하게 된다. 이러한 루프에 대해서는 FES 기법은 올바른 동기화를 수행하지 못하게 된다. 따라서 두가지 이상의 상태를 표현하기 위해 동기화 변수를 one-bit key 대신에 정수 타입의 동기화 변수로 확장한 것이 EFE(Extended Full/Empty) 기법이다.

이 기법을 이용해서 동기화를 시킨 재구조화 루프는 그림 5의 (b)와 같다.

```
do i = 3, N
S1 : A[i]  = . . .
S2 : .  . = A[i-2]
S3 : A[i-1] = . . .
end do
```

(a) 예제 루프

```
do_all i = 3, N
T1 : A[i]%KEY = 0
end do_all
T2 : A[1]%KEY = 2
T3 : A[2]%KEY = 1
do_across i = 3, N
W1 : with A[i] when (A[i]%KEY = 0)
S1 :   A[i]%DATA = . . .
       call wait_for_memory()
       A[i]%KEY = 1
     end with
W2 : with A[i-2] when (A[i-2]%KEY = 2)
S2 :      . . . = A[i-2]%DATA
S2a :  A[i-2]%KEY = 3
     end with
W3 : with A[i-1] when (A[i-1]%KEY = 1)
S3 :   A[i-1]%DATA =
       call wait_for_memory()
       A[i-1]%KEY = 2
     end with
end do_across
```

(b) 동기화된 루프

그림 5 확장된 Full/Empty 동기화 기법

그림 5에서 사용된 with 문장의 의미는 다음과 같다.

```
with resource when test
    statement list
end with
```

여기서, statement list는 임계 영역으로 임계 영역의 수행은 test가 참이 될때까지 대기 상태에 있게 되고, resource는 스칼라 변수, 배열 원소 또는 구조체 변수를 의미하며 test는 resource의 값 또는 상태에 대한 부울린 표현식을 나타낸다.

(2) 문장 지향 기법

문장 지향 기법은 문장의 올바른 수행 순서를 유지하기 위해서, 각 문장 $S_p(S_p \delta S_q)$는 하나의 동기화 변수, SC(Statement Counter)를 할당받는다. 이 동기화 변수는 source인 S_p의

모든 인스턴스들에 의해서 공유된다.

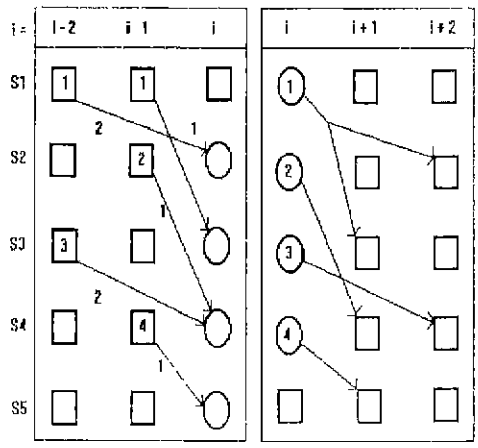
이 기법은 다음과 같은 방법으로 실행 순서를 유지한다. 반복 i가 문장 S_p의 수행을 완료한 후 SC의 값을 i로 증가하기 전에 SC = i−1이 될 때까지 대기하게 한다. 따라서 SC = i−1일 때 모든 반복 j(j≤i)는 S_p의 수행을 완료한 상태가 된다. 초기에 첫번째 반복이 k라면 SC는 k−1의 값을 갖는다. 반복 i가 sink 문장 S_q를 수행하기 위해서는 이전의 해당 source 문장의 수행이 SC≥i−D 인가를 점검해야 한다(여기서 D는 $S_p \rightarrow S_q$의 거리를 의미).

그림 6은 그림 3의 (a)에서 제시한 루프에 대한 문장 지향 동기화 수행 방법을 나타낸 것이며, 그림 6의 (b)에서, 종속 관계를 갖는 모든 문장은 하나의 동기화 변수를 공유하기 때문에 문장간의 종속 관계에 따라 순차적으로 동기화 변수의 상태를 수정한다. 또한 (a)에서는 동기화 변수의 값과 자신의 종속 거리에 따라 수행 여부를 결정한다.

이러한 형태의 동기화 기법으로는 Advance &Await[13], Fetch&Add[14], Post&Wait [6], [29], Signal&Wait[21], Test&Set[11], Test&Testset[20] 등이 있으며, Test/Testset 동기화 명령어에 대한 간단한 설명은 다음과 같다.

Test 명령은 두개의 인수를 갖는다. 하나는 테스트하고자 하는 동기화 변수이고 다른 하나는 종속 거리를 의미한다. test 명령은 동기화 변수의 값이 동기화해야 할 source 데이타를 포함하고 있는 루프 인덱스의 값과 적어도 동일할 때까지 실행을 대기상태로 유지한다. 예를 들면, 어떤 작업 수행을 위해서 이전에 두개의 반복의 수행이 완료되어야 하는 경우에 Test (R, 2) 동기화 명령이 생성된다. 반복 i에서 test 명령은 변수 R의 값이 적어도 i−2가 될때까지 기다리게 된다(그림 7의 (a)참조).

Testset 명령은 단지 하나의 인수를 갖는다. Testset 명령은 어떤 사건이 발생했음을 나타낸다. 이전의 반복에서 testset 명령이 수행되었는 가를 확인하기 위해서 동기화 변수의 값을 검사한다. 즉, 변수의 값이 이전 인덱스의 값과 동일한 가를 확인함으로써 위의 작업을 수행한다. Testset 명령이 이전 반복에서 수행되었을때 testset은 현재 인덱스의 값으로 변수의 값을 변경한다. Testset 명령은 인수의 값을 순서적으로 수정해야 하기 때문에 병렬성을 감소시킨다(그림 7의 (b) 참조).



(a) sink 수행 (b) source 수행

Ⓝ——> = Advance(N)
= wait until SC[N] = i-1;
set SC[N] to i

Ⓝ —d—> ○ = Await(d,N)
= wait until SC[N]≥i-d

그림 6 문장 지향 동기화 기법의 수행 방법

```
Test(R, △)                      Testset(R)
  with R when I - △ <= R          if (R < 1) then
  end with                          with R when (R=i-1)
                                        R = R + 1
                                    end with
                                    end if
```

(a) test 명령 (b) testset 명령

여기서, R : 루프의 초기에 0의 값는 동기화 변수
△ : 종속 거리
I : 루프의 인덱스

그림 7 test/testset 명령어의 의미

Testset/Test 동기화 명령어를 종속 관계를 갖는 문장에 삽입하기 위한 알고리즘은 그림 8과 같다.

```
foreach dependence δ_i in the loop do
    Δ = GCD(Greatest Common Divisor) for the distance set of δ_i
    R_i = a compiler-generated integer variable
    if δ_i is a LFD(Lexically Forward Dependence) then
        add the instruction testset(R_i)
            immediately after source(δ_i)
    else
        add the instruction test(δ_i, Δ)
            immediately before sink(δ_i)
        add the instruction testset(R_i)
            immediately after source(δ_i)
    end if
end foreach
```

그림 8 test/testset 명령어를 이용한 루프의 동기화를 위한 알고리즘

이 동기화 명령어를 이용하여 그림 9의 (a)를 동기화 시킨 예는 (b)와 같다.

그림 7의 (b)에서 보는 바와 같이 종속 관계를 갖는 문장은 하나의 동기화 변수(R)을 공유하기 때문에 데이타 지향 기법과는 달리 동기화 변수의 사용으로 인한 오버헤드는 존재하지 않지만, (c)에서 보는 바와 같이 i = 2, 3, 4, ... 상의 문장 S2는 S1의 수행이 완료된 후 바로 수행되어야 하지만 동기화 명령(testset)에 의해 결과적으로 순차적으로 수행하게 된다.

```
do I = 1, N
S1 : A[I] = B[I-2] + C[I]
S2 : B[I] = A[I] + D[I]
end do
```

(a) 예제 루프

```
do_across I= 1, N
T1 : tets(R,2)
S1 : A[I] = B[I-2] + C[I]
S2 : B[I] = A[I] + D[I]
TS1 : testset(R)
end do_across
```

(b) 동기화된 재구조화 루프

i =	1	2	3	4
time ↓	T1	T1	...	...
	S1	S1	...	...
	S2	S2	...	...
	TS1	...	T1	...
		TS1	S1	**T1**
			S2	S1
			TS1	S2
				TS1

(c) 수행 순서

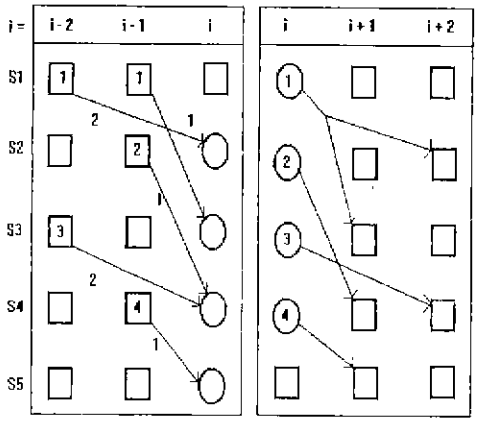
그림 9 testset/test 동기화 기법

(3) 프로세스 지향 기법

프로세스 지향 기법[27][8]은 문장 지향 기법과 많은 유사한 점을 내포하고 있다. 이 기법의 기본적인 방법은 각각의 데이타 또는 문장에 동기화 변수를 할당하는 것이 아니라 각각의 반복(또는 프로세스)에 동기화 변수, PC (Process Counter)를 할당하는 것이다. PC는 프로세스 자체에 의해서만 수정될 수 있는 공유 변수로써 두가지 정보(process id, step)을 포함하고 있는 한 프로세스 상태의 일부로 간주된다. 한 PC의 step은 각 source 문장의 수행이 완료된 후 수정된다(그림 10의 (b) 참조). 따라서 PC의 최대 step은 각 반복내의 source 문장의 전체수가 된다. 또한 sink 문장의 수행은 모든 해당 source 문장의 PC를 점검한 후 수행 여부를 결정한다(그림 10의 (a) 참조).

프로세스 지향 동기화 기법을 이용하여 루프를 동기화하기 위한 몇가지 동기화 연산은 그림 11와 같다.

그림 11에서 각 동기화 연산의 의미는 다음과 같다. (1) set_PC는 source 문장의 수행이 끝난 후 PC의 step를 수정. (2) release_PC는 마지막 source 문장의 수행이 끝난 후 PC를 다



(a) sink 수행 (b) source 수행

Ⓝ——> ≡ set_PC(N)

set PC[i] to N

Ⓝ —d—> ○ ≡ wait_PC(d,N)

≡ wait until PC[i-d]≥i-d

그림 10 프로세스 지향 동기화 기법의 수행 방법

```
i : 수행하고자 하는 인덱스 변수의 값
X  PC의 수
mod  modulus operation
PC의 형식  <owner, step>
          <w,x> ≥ <y,z> iff w > y or w = y and x≥ z
초기화, PC[i] = <i,0>  1 ≥ i ≥ X
· set_PC(current_step)  /* update PC to current step */
  PC[i mod X].step ← current_step
· release_PC()  /* release PC for process i+X to use */
  PC[i mod X] ← <i+X,0>
· wait_PC(dist, step)  /* i-dist : pid of the source */
  while(PC[i-dist] mod X] < <i-dist,step>)
· get_PC()  /* get the ownership of PC */
  wait_PC(0, 0)
```

그림 11 프로세스 지향 기법의 동기화 연산

음 프로세스에 전달. (3) wait_PC는 sink 문장을 수행하기 전에 해당 source 문장의 수행이 완료될 때까지 대기 상태로 유지. (4) get_PC는 PC의 ownership을 얻기 위한 대기 상태를 의미한다.

그림 3을 이 기법을 이용하여 재구조화시킨 루프는 그림 12와 같다.

```
do_across i = 1, N
 S1(i).
 get_PC().        /* wait for the PC to be available */
 set_PC(1).       /* completion of source 1 */
 wait_PC(2,1)     /* until process i-2 completes source 1 */
 S2(i).
 set_PC(2);
 wait_PC(1,1).
 S3(i).
 set_PC(3);
 wait_PC(1,2).
 wait_PC(2,3).
 S4(i)
 release_PC().    /* complete last source and release PC */
 wait_PC(1,4).
 S5(i)
end do_across
```

그림 12 그림 3에 대한 재구조화된 루프

3.1.2 가변 종속거리 루프

앞에서 제시한 불변 종속거리 동기화 기법은 각 인스턴스간에 일정한 형태의 종속성을 갖는 루프에 대해서는 효율적으로 동기화를 수행할 수 있지만, 인스턴스간에 가변 종속거리가 존재하는 경우에는 종속 관계가 없는 인스턴스에도 동기화 정보를 유지해야 하기 때문에 불필요한 동기화 변수 및 동기화 명령으로 인한 오버헤드가 발생하여 효율적으로 동기화를 수행할 수 없다.

배열 변수의 첨자식과 데이타 종속성에 대한 연구[25][28]에 의하면 데이타 종속 형태에서 작은 부분(13.65%)만이 불변 종속거리를 갖는다. 즉, 86.35%가 가변 종속거리를 갖는다. 따라서 병렬처리를 위한 효율적인 동기화를 수행하기 위해서는 가변 종속거리가 존재하는 루프에 대해서도 적용할 수 있어야 한다.

(1) RDC(Run-time Dependence Checking)

RDC(Run-time Dependence Checking)[22]의 기본적인 방법은 루프내의 특정 반복이 하나 또는 그 이상의 반복에 종속이 되는가를 실행 시에 결정한다. 이에 따라 동기화가 필요한 반복에 대해서만 동기화 명령을 수행함으로써 가변 종속 거리 루프를 병렬처리할 수 있다.

RDC 알고리즘은 다음과 같다.

RUN-TIME DEPENDENCE CHECKING
▣ IMPLICIT PHASE
□ STEP 1 :
각각의 정적 종속 루프 변수 i에 대해, source vector, $R_i(1:N)$를 생성.
컴파일러가 다음의 루프를 수행.
DOALL I = 1, N
$R_1(e_1(i)) = I : V_1(e(i)) = 1$
$R_2(e_2(i)) = I : V_2(e(i)) = 1$
. . .
$R_n(e_n(i)) = I : V_n(e(i)) = 1$
ENDD
여기서, $e_j(i)$는 루프 변수가 i일때, j번째 source 정적 종속의 첨자식의 값을 의미.

▣ EXPLICIT PHASE
□ STEP 2
j번째 정적 종속에 대해서 해당 루프의 첫부분에 다음의 문장을 삽입.
IF $(1 \le R_j(h_j(i)) < I)$ THEN WAIT ON $V_j(h_j(i))$
여기서, $h_j(i)$는 sink의 첨자식의 값을 의미.

□ STEP 3
해당 루프의 끝부분에 다음의 문장을 삽입.
CLEAR $V_j(e_j(i))$

위의 동기화 알고리즘을 이용하여 재구조화 시킨 루프는 그림 13과 같다.

```
Do i = 1, N
  A[2i-1] = B[i-1] + 1
  B[2i+1] = A[i+1] × C[i]
End do
```

(a) 순차 루프

```
Do_all i = 1, N
  IF (1 ≤ R1(i+1) < i) WAIT ON V1(i+1)
  IF (1 ≤ R2(i-1) < i) WAIT ON V2(i-1)
  A[2i-1] = B[i-1] + 1
  B[2i+1] = A[i+1] × C[i]
  CLEAR V1(2i-1)
  CLEAR V2(2i+1)
End Do_all
```

(b) 재구조화 루프

그림 13 RDC 기법에 의한 재구조화

(2) Synch-Read/Synch-Write

[28]에서는 루프의 각 데이타의 모든 인스턴스에 대한 순차 접근 순서(sequential access order)를 이용하여 데이타 종속성을 위한 동기화 기법(synch-read/synch-write)를 제시하였다. 이 기법은 논문 [30]에서 제시한 방법과 비슷하지만, 순차 접근 순서를 계산-접근 방법(computational approach)를 이용함으로써 연산-접근 방법(operational approach)을 이용하는 위 기법의 문제점인 실행 시에 전역 공유 테이블(global shared table)과 장벽 동기화 변수를 접근하는데 소요되는 시간 지연을 최소화 하였다.

이 기법에 대한 동기화 알고리즘은 다음과 같다.

```
▣ synch-read(x, s)
  with x when KEY > s
       increment the KEY
       read x
  end with

▣ synch-write(x, s)
  with x when KEY = s
       increment the KEY
       write x
  end with
```

여기서 x는 동기화 변수의 이름(또는 주소)을 의미하고, s는 프로세서가 Synch-Read 또는 Synch-Write를 수행하면서 제공되는 정수(integer)를 의미한다.

3.2 장벽 동기화 기법(Barrier Synchronization Scheme)

장벽 동기화 기법[29]은 세마포어(sema-

phore)와 같이 병렬 프로세서간의 접근 경쟁을 해결하는데 쓰이는 것이 아니라, 한 작업을 병렬로 수행하는 프로세스들의 종료 시간을 조절하는데 쓰인다.

이 기법은 루프의 여러 지점에 장벽(barrier)을 두어 모든 반복들이 장벽을 만날 때까지 어떤 프로세서도 장벽 다음의 문장들을 수행할 수 없게 하는 기법이다. 즉, 루프내에 장벽을 둠으로써 루프를 몇개의 세그먼트들로 나누고, 모든 반복들에서 다음 세그먼트의 실행 전에 현재의 세그먼트의 실행을 완료하게 한다.

장벽 동기화는 문장간의 데이타 종속이 순차적으로 위에서 아래로의 전진 종속(forward dependence)의 경우에만 허용된다. 따라서 루프를 종속 관계가 없는 블럭 단위로 나누어 부분 병렬화를 꾀할 경우에 종종 쓰인다.

루프내에서 각 반복마다 각각의 프로세서를 할당한다면, 그 루프의 수행 시간은 가장 늦게 할당된 반복을 수행하는 프로세서에 의존적이다. 현재 수행하고 있는 블럭내의 모든 반복들이 완료되기 전에는 다음 블럭을 수행할 수 없기 때문에 만일 가장 늦은 프로세스의 수행 시간이 평균 수행 시간보다 훨씬 길다면 나머지 대부분의 프로세스들은 가장 늦은 프로세스가 수행하는 시간의 대부분을 휴식 상태(idle)에 있게 된다. 따라서 장벽 동기화는 모든 프로세스들의 수행 시간이 짧거나 혹은 거의 같은 시간에 수행이 종료된 경우에 유용하다고 할 수 있다.

3.3 파이프라인 동기화 기법(Pipeline Synchronization Scheme)

임의 동기화 기법은 가장 융통성이 있는 동기화 기법이지만 이 기법을 최적화하는 것은 특정 컴퓨터 구조의 제약성때문에 매우 어렵다. 동기화 기법의 제약 조건을 둠으로써 좀 더 쉽게 동기화를 구현할 수 있는 기법으로 파이프라인 동기화 기법[29]이 있다.

파이프라인 동기화 기법은 루프를 몇개의 세그먼트(segment)들로 나누고, 파이프라인 구조의 산술 장치와 같이 이 세그먼트들을 중첩(overlap)시켜 루프를 수행한다. 이때 생성된 세그먼트들에 대해 모든 데이타 종속은 동일한 세그먼트에 존재하거나 다음 세그먼트로 나누어진다. 모든 세그먼트들의 실행 시간을 동일하게 하기 위해 가장 큰 세그먼트가 모든 세그먼트들의 크기가 된다.

```
do i = 1, N
  Seg1 : S1   A[i] = B[i] + C[i]
  Seg1 : S2 : C[i] = A[i-1] + C[i]
  Seg2 : S3 : D[i] = A[i] × 2
  Seg2 : S4 : E[i] = D[i] + C[i-2]
  Seg3 : S5 : F[i] = E[i] + F[i]
  Seg3 : S6 : G[i] = SQRT(F[i])
end do
```

그림 14 파이프라인 동기화 기법의 수행 방법

그림 14에서, 문장 S1의 배열 변수 A[i]와 S_2의 A[i−1]는 후향(backward) 종속 관계가 성립하고($S_2\delta S_1$), S_1의 배열 변수 C[i−1]과 S_2의 C[i]는 전향(forward) 종속 관계가 성립한다($S_1\delta^a S_2$). 따라서 두개의 문장 S_1과 S_2는 동일한 세그먼트에 위치해야 한다. 즉, 하나의 세그먼트의 길이는 두개의 문장으로 구성된다. 따라서 이 길이는 전체 루프의 세그먼트의 크기가 된다.

그림 14의 루프 실행 과정은 그림 15와 같다.

```
루프변수 : i =1     i = 2     i = 3     i = 4
          Seg1
          Seg2      Seg1
          Seg3      Seg2      Seg1
                    Seg3      Seg2      Seg1
                              Seg3      Seg2
```

그림 15 파이프라인 동기화 기법의 수행 방법

임의 동기화 기법에 비해 파이프라인 기법은 동기점(synchronization point)들의 수를 줄

일 수 있다는 장점이 있다. 그러나 동기화될 필요가 없는 코드를 중첩 수행량을 증가시키기 위해 별개의 세그먼트들로 나눔으로써 더 많은 동기화를 유발할 수 있다. 예를 들면 그림 15에서 문장 S_4와 S_5 사이에는 동기화를 수행할 필요가 없지만, 두 문장사이의 동기점이 없다면 이 루프는 두개의 세그먼트로 구성되고 최대 중첩 정도는 2가 되어 병렬 수행의 정도가 낮아지게 된다. 또다른 단점은 만약 루프내의 모든 문장이 종속성 사이클(cycle)을 형성하게 되면 전체 루프가 동일한 세그먼트내에 존재해야 되기 때문에 결과적으로 순차처리할 수 밖에 없다. 또한 후향 종속성이 존재하지 않는 루프에 대해서도 이 기법은 수행할 수 있는 병렬성 정도에 제약을 받게 된다.

3.4 임계 영역(Critical Section)을 이용한 동기화 기법

임계 영역을 이용한 동기화 기법[29]은 데이타 종속 관계가 있는 루프에 임계 영역을 삽입하고, 그 영역은 한번에 한 프로세서에 의해서만 실행되도록 하여 병렬 실행을 동기화시키는 방법이다. 이때 이 임계 영역의 크기는 가능한 한 작게하여야 함으로 여러가지 루프 변환 알고리즘을 이용하여 독립된 문장들은 따로 묶음으로써 임계 영역의 크기를 최소화할 수 있다. 이 기법은 후향 종속성을 갖는 루프를 처리할 수 있고, 임계 영역이외의 코드들은 파이프라인 동기화 기법과는 달리 독립적으로 실행할 수 있다는 장점이 있다.

```
do i = 1, N
     Begin Critical Section
S1 : A[i] = A[i-1] × B[i] + C[i]
     End Critical Section
S2 : C[i] = A[i] + C[i]
S3 : D[i] = A[i] × 2
S4 : E[i] = D[i] + C[i]
end do
```

그림 16 임계 영역 기법에 의한 재구조화 루프

그림 16에서, 문장 S_1은 자체적으로 데이타 종속 사이클을 형성하기 때문에 문장 S_1만을 임계 영역에 삽입시킴으로써 루프의 병렬 수행을 유지할 수 있다. 따라서 프로세서가 임의의 시간에 문장 S_1을 수행한 후 루프의 나머지 문장들은 독립적으로 수행할 수 있게 된다.

4. 결 론

본 고에서는 공유 메모리를 갖는 MIMD 시스템상에서 루프의 병렬 처리를 위한 동기화 기법에 대해서 살펴보았다.

앞에서 제시한 동기화 기법은 루프의 구조에 연구의 촛점을 두고 있다. 그 이유는 일반적으로 루프의 구조가 전체 수행 시간 중 상당부분을 차지하는 자원이고 가장 중요한 병렬성 추출의 기본이 되기 때문이다.

동기화 기법은 동기화 수행 방법 및 제약성에 따라 임의 동기화, 장벽 동기화, 파이프라인 동기화 및 임계 영역을 이용한 동기화 기법으로 나눌 수 있다. 이 중에서 임의 동기화 기법은 다른 기법에 비해 병렬성 추출의 정도가 가장 높고 융통성이 많기 때문에 가장 많은 연구가 진행되는 기법이다. 이에 따라 본 논문에서도 임의 동기화 기법에 대해서 좀 더 자세하게 다루었다.

앞으로 병렬 컴퓨터가 대중화되고 있는 시점에서 이를 효과적으로 수행하기 위해서는 새로운 프로그래밍 환경, 병렬 언어의 구성 등 많은 관련 연구가 진행되어야 하지만 무엇보다도 기존의 응용 프로그램을 병렬 컴퓨터상에서 쉽게 구현하기 위한 재구조화 컴파일러의 연구가 선행되어야 할 것이다.

참 고 문 헌

[1] A. Aiken, A. Nicolu, "Optimal Loop Parallelization, Proc. of the SIGPLAN '88 Conference on Programming Language Design and Implementation", ACM pp.308-317, 1988.

[2] J. R. Allen, D. Baumgrrtner, K. Kennedy, A. Porterfield, "PTOOL : A Semi-Auto-

matic Parallel Programming Assistant", IEEE, pp.164-170, 1986.

[3] Alliant FX/Series Architecture Manual, Alliant Computer Systems Corp. 1989.

[4] U. Banerjee, Dependence Analysis for Supercomputing, Kluwer Academic Pub. 1988.

[5] BBN Advanced Computers Inc., "Programming in Fortran with the Uniform System", Cambridge, MA.

[6] P. Callahan, K. Kennedy, J. Subhlok, "Analysis of Event Synchronization in Parallel Programming Tool", ACM ICS, pp.21-30, 1990.

[7] H. Cheng, "Vector Pipelining, Chaining and Speed on the IBM 3090 and Cray X-MP", IEEE Comp. pp.31-46, Sep. 1989.

[8] T. Chen, C. Q. Zhu, "A New Synchronization Mechanism", Intl. Conference on Parallel Processing, pp.176-179 Aug. 1991.

[9] M. Cosnard, K. Ebcioglu, J. Gaudiot, "Architectures and Compilation Techniques for Fine and Medium Grain Parallelism", North-Holland, 1993.

[10] A. Das, L. Moser, P. Mellear-Smith, "PAL : A Language for Parallel Asynchronous Computation", Proc. of Intl. Conference on Parallel Processing, pp.166-173, Aug. 1992.

[11] A. Dinning, "A Survey of Synchronization Methods for Parallel Computers", IEEE Computer, pp.66-77, July 1989.

[12] R. Eigenman, J. Hofflinger, G. Jaxon, D. Padua, "Cedar Fortran and Its restructuring Compiler", CSRD Report No. 1041, Center for Supercomputer Rsearch and Development, Univ. of Illinois, Sept. 1991.

[13] P. Emrath, D. Padua, "Automatic Detection of Non-determinancy in Parallel Programs", Proc. of ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging, pp.89-99, 1988.

[14] A. Gottlieb, R. Grishman, C. P. Kruskal, K. P. MaAuliffe, "The NYU Ultracomputer-Designing on MIMD Shared Memory Parallel Computer", IEEE Tran. on Computer, Vol. C-32, No. 2, Feb. 1983.

[15] C. Houck, G. Agha, "HAL : A High-Level Actor Language and Its Distributed Implementation", Proc. of Intl. Conference on Parallel Processing, pp.158-165, 1992.

[16] V. P. Krothapalli, P. Sadayappan, "An Approach to Synchronization for Parallel Computing", Intl. Conference on Supercomputing, pp.4-8, July 1988.

[17] D. J. Kuck et al., "The Effect of Program Restructuring, Algorithm Change and Architecture Choice on Program Performance", In Proc. Int. Conf. on Paralle Processing, pp.129-138, Aug. 1984.

[18] J. Kuck, E. S. Davison, D. Lowrie, A. H. Sameh, "Parallel Supercomputing Today and Cedar Approach", Science, Vol. 231, pp.967-974, Feb. 1986.

[19] A. Malony, J. Larson, D. Read, "Tracing Application Program Execution on the Cray X-MP and Cray-2", CSRD Report No. 985, CSRD Univ. of Illinois at Urbana-Champaign, Nov. 1990.

[20] S. P. Midkiff, D. Padua, "Compiler Algorithms for Synchronization", IEEE Tran. on Computer Vol. C-36, No. 12, pp. 1485-1495, Dec. 1987.

[21] D. Padua, M. J. Wolfe, "Advanced Compiler Optimization for Supercomputers", Communication of ACM, Vol. 29, No. 12, pp.1184-1201, Dec. 1986.

[22] C. D. Polychronopoulos, "More on Advanced Loop Optimization", CSRD Report No. 667, CSRD at Univ. of Illinois, Oct. 1987.

[23] K. Psarris, "On Exact Data Dependence Analysis", Intl. Conference on Supercomputings, pp.303-312, July 1992.

[24] J. A. Sharp, Data Flow Computing, Ellis Horwood Limited Press, 1985.

[25] Z. Shen, Z. Li, P. C. Yew, An Empirical Study on Array Subscripts and Data Dependencies, Proc. of the Intl. Conference on Parallel Processing, pp.145-152, Aug. 1989.

[26] B. J. Smith, A Pipelined, Shared Resource MIMD Computer, Proc. of the Conference

on Parallel Processing, Aug. 1978.

[27] H. M. Su, P. C. Yew, "On Data Synchronization for Multiprocessor", Proc. of the Annual Intl. symposium on Computer Architecture, pp.416-423, May 1989.

[28] P. Tang, P. C. Yew, C. Q. Zhu, "Compiler Techniques for Data Synchronization in Nested Parallel Loops", Proc. of the ACM Intl. Conference on Supercomputing, pp. 176-181, July, 1990.

[29] M. E. Wolf, "Multoprocessor Synchronization for Concurrent Loops", IEEE Software, pp.34-42, 1988.

[30] C. Q. Zhu, P. C. Yew, "A Scheme to Enforce Data Dependence on Large Multiprocessor System", IEEE Tran. on Software Engineering Vol. SE-13, No. 6, pp. 726-739, June 1987.

박 두 순

1981 고려대학교 수학과 졸업(학사)
1983 충남대학교 계산통계학과 졸업(석사)
1988 고려대학교 대학원(전산학 전공) 졸업(박사)
1992~1993 미국 Univ. of Illinois at Urbana-Champaign CSRD 객원교수
1985~현재 순천향대학교 전산학과 부교수
주관심분야 : 병렬처리 컴파일러, 계산이론, 프로그래밍 언어론 등임

이 광 형

1988 순천향대학교 전산학과 졸업(학사)
1991 고려대학교 전산과학과 대학원 졸업(석사)
1995 고려대학교 전산과학과 대학원 졸업(박사)
주관심분야 : 병렬/분산처리 시스템, 컴파일러, 프로그래밍 언어론 등임

황 종 선

1978 Univ. of Georgia, Dept. of Stat. & Computer Science 박사
1978~1980 미국 South Carolina 주립대학교 조교수
1980~1981 미국 상무성 연방표준국 연구위원
1981~1982 한국표준연구소전자계산실장
1986~1989 한국정보과학회 부회장
1982~현재 고려대학교 전산과학과 교수
1995~현재 한국정보과학회 회장
주관심분야 : 병렬처리 알고리즘, 분산처리, 인공지능론 등임

김 병 수

1981 서울대학교 수학과 졸업(학사)
1987 미국 오클라호마 주립대 전산학과 졸업(석사)
1993 고려대학교 전산과학과 대학원 졸업(박사)
1987~1991 한국과학기술원 시스템공학연구소 근무
1991~현재 건양대학교 전자계산학과 조교수
주관심분야 : 병렬처리, 컴파일러, 알고리즘 등임