Java 프로그래밍에서 단일 루프 구조의 병렬성 검출

황 득 영[†] · 권 오 진[††] · 최 영 근[†††]

요 약

순차 Java 프로그램을 병렬 시스템에서 실행할 경우 루프는 전체 수행 시간 중 많은 부분을 차지하므로 병렬성 검출의 기본이 된다. 본 논문은 기존에 작성된 단일 루프 구조를 갖는 Java 프로그래밍 언어에서 종속성 분석을 수행하여 묵시적 병렬성을 검출하는 방법을 제안한다. 또한 재구성 컴파일러에 의하여 병렬 코드를 생성하는 방법과 Java 원시 프로그램을 Java 프로그래밍 언어 자체에서 지원하는 다중스레드 기법으로 변환하는 방법을 제안한다. 스레드 문장으로 변환된 프로그램에 대해 루프의 반복계수와 스레드 수를 매개변수로 하여 성능 분석을 하였다. 재구성 컴파일러에 의한 장점은 사용자의 병렬성 검출에 대한 오버헤드를 줄이고, 순차 Java 프로그램에 대한 효과적인 병렬성 검출을 가능하게 한다.

Exploiting implicit Parallelism for Single Loops in Java Programming Language

Deuk-Young Hwang[†] · Oh-Jin Kwon[††] · Young Gun Choi[†††]

ABSTRACT

The loop is a fundamental for exploiting parallelism as it has a large portion of execution time for sequential Java program on the parallel machine. This paper proposes the method of exploiting the implicit parallelism through the analysis of data dependence in the existing Java programming language having a single loop structure. The parallel code generation method through the restructuring compiler, and the method of translating Java source program into multithread statement, which is supported in the level of the Java programming language, are proposed here. The performance test of the program translated into the thread statement is conducted using the trip count of loop and the thread count as parameters. The restructuring compiler makes it possible for users to reduce overhead and exploit parallelism efficiently in the Java programming.

1. 서 론

정보의 처리 내용이 고도화 및 다양화됨에 따라 최근 병렬처리는 객체지향의 개념이 도입되어 연구되고 있다. 절차형 언어에서 데이터의 흐름은 제어의 흐름과 독립되어 있는 반면 객체지향은 데이터와 제어가 함께 이동하는 메소드 호출로서 연산이 이루어진다. 즉, 객체지향은 모듈성이 강하고 모듈 상호간에 메시지를 통해서 연산을 수행하며 객체지향 개념 자체에 병행성(concurrency)을 포함하고 있다. 따라서 객체지향의 개념은 병렬처리에 적합하며 실세계의 개체를 하나의 객체로 모델링할 수 있으므로 문제를 기술하는 것만으로 다수의 계산 모듈을 생성하게 된다. 즉 하나의 객체

※ 이 논문은 1998년도 광운대학교 학술연구비에 의하여 연구되었음.
† 준 회 원 : 삼척대학교 컴퓨터과학과
†† 정 회 원 : 산업기술정보원 데이터베이스사업부 연구원
††† 정 회 원 : 광운대학교 전자계산학과
논문접수 : 1998년 3월 14일, 심사완료 : 1998년 4월 29일

를 하나의 프로세스로 모델링하게 되면 한 시스템 내부에서는 여러 프로세스가 동시에 존재하게 되며, 이러한 객체지향 모델은 병렬 프로그램을 위한 모델로 자연스럽게 이용될 수 있다[5,10].

병렬 프로그래밍 언어의 유형은 크게 두 가지로 분류할 수 있다. 첫째는 자원관리 및 특정 기계의 세부사항을 명시하는 명시적인 병렬 언어에 의한 접근 방법이고, 둘째는 순차 프로그램을 입력받아 병렬 프로그램으로 변환하는 병렬화 컴파일러에 의한 방법으로 분류한다.

명시적인 병렬 언어는 순차 프로그래밍 언어에 병렬 프리미티브(primitive) 추가에 의해 명시적인 병렬 언어를 만드는 방법으로 여기에 포함되는 언어는 Concurrent Pascal, MultiLisp, QLisp, Concurrent Prolog, PARLOG 등이 있다. 그러나 명시적인 병렬 언어는 새로운 병렬 프로그래밍 언어를 습득해야 하는 오버헤드가 있고, 병렬성의 검출을 사용자에게 요구하는 단점을 가지고 있다. 기존에 작성된 많은 Java 프로그램을 병렬 프로그램으로 변환하는 데에는 객체지향의 개념을 사용하였다 할지라도 많은 시간이 소모된다.

병렬화 컴파일러에 의한 방법에서 대부분의 순차 프로그램들은 루프를 처리하는데에 많은 시간을 소요하므로 속도 향상을 위해 프로그램을 변환시켜 주는 작업이 필요하다. 이러한 기능을 하는 자동 변환 컴파일러(restructuring compiler)는 PARAFRASE, KAP, PFC, PTRAN, BULLDOG 등이 있으며 이 컴파일러들은 기존의 순차 프로그램에서 제어 종속성(control dependence) 및 자료 종속성(data dependence)을 분석하고, 이 과정에서 검출한 정보를 이용하여 프로그램을 벡터화(vectorization) 또는 병렬화(parallelization)하는 연구가 다양하게 진행되어 왔다. 병렬화 컴파일러에 의한 방법은 프로그래머가 병렬성을 명시하거나 찾아낼 필요가 없다는 장점이 있다[2,3,9].

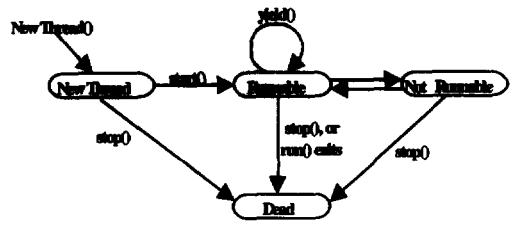
따라서 기존의 순차적 프로그램을 재사용할 수 있기 위해서는 다양한 모델의 병렬 시스템에서 사용할 수 있도록 프로그램 내에 존재하는 병렬성을 찾아내는 것이 중요하다. 일반적으로 응용 프로그램에서 루프는 전체 수행 시간 중 많은 부분을 차지하는 자원이고 가장 중요한 병렬성 검출의 기본이 된다. 루프의 형태는 모든 반복주기(iteration)들 사이에 종속성이 없는 경우에 병렬 구문 DOALL로 변환하거나 서로 다른 반복주기 사이에 종속성이 발생하는 경우에 DOACROSS로 변환하여 병렬 처리하는 방법이 있다[2,3,16].

본 논문은 기존에 작성된 다중 단일 루프 구조를 갖는 Java 프로그래밍 언어에서 데이터 종속성 분석을 수행하여 묵시적(implicit) 병렬성을 검출한다. 또한 재구성 컴파일러에 의하여 병렬 코드를 생성하는 방법과 Java 원시 프로그램을 Java 프로그래밍 언어 자체에서 지원하는 다중스레드(multi-thread) 기법으로 변환하는 방법을 제안한다. 그리고 제시한 방법을 다중스레드 기법에 적용하여 여러 가지 측면에서 비교·분석한다. 재구성 컴파일러에 의한 이점은 사용자의 병렬성 검출에 대한 오버헤드를 줄이고, 순차 Java 프로그램에 대한 효과적인 병렬성 검출을 가능하게 한다.

2. Java 언어에서 다중스레드 및 데이터 종속성

2.1 스레드

스레드는 프로세스 스케줄링의 부담을 줄여 성능을 향상시키기 위한 프로세스의 다른 표현 방식이다. 스레드는 보통 경량(lightweight) 프로세스 또는 실행문맥이라고도 하는데, C와 C++언어는 단일 실행 스레드에 속하는 언어로서, 스레드에 대한 언어 수준(language-level) 지원을 제공하지 못한다[4,6,8]. 하지만 Java에서는 언어 차원에서 하나의 주소 공간(address space)을 공유하는 여러 개의 스레드들이 병행적으로 실행할 수 있는 다중스레드를 지원한다. 본 논문에서는 병렬 프로그램의 각 프로세서와 제어 흐름을 나타내기 위해 다중스레드를 사용한다. (그림 1)은 어떤 특정 시점에서의 Java 스레드 상태를 나타낸다[11].



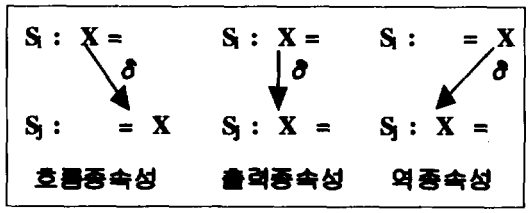
(그림 1) 스레드 전이 상태
(Fig. 1) Transition diagram for Thread

(그림 1)의 New Thread 상태는 시스템의 자원을 아직 할당하지 않은 공백 스레드 객체를 나타내고, start()와 stop() 메소드를 호출하여 스레드를 시작하거나 중지할 수 있다. Runnable 상태는 start() 메소드에 의하여 스레드를 실행하기 위하여 필요한 시스템 자원을 생성하고, 스케줄링을 한 후 run() 메소드를 호출한다. 그러나 Runnable이 Running을 표현하는 것은 아니므로, 단일 프로세서를 갖는 시스템에서는 동시에 모든 Runnable 스레드를 실행할 수 없기 때문에 Java 런타임 시스템은 모든 Runnable 스레드를 프로세서에게 분배하는 스케줄링 방법을 수행해야 한다. Not Runnable 상태는 sleep()이나 suspend()메소드를 호출할 때, 조건 변수를 만족하기 위하여 wait() 메소드에 의하여 대기 상태일 때, 스레드가 I/O를 블록킹(blocking)할 때 발생한다. 스레드가 Not Runnable 상태를 벗어나기 위해서 sleep()에서는 잠자는 시간을 초과하거나, suspend()에서는 resume() 메소드를 호출할 때, wait()에서는 notify()나 notifyAll() 메소드를 호출할 때, I/O 블록에서는 I/O를 끝냈을 때 해제된다. Dead 상태는 stop() 메소드를 호출하거나, run() 메소드가 정상적으로 종료할 때 발생한다.

Java에서 새로운 스레드를 얻기 위한 가장 간단한 방법은 java.lang.Thread의 서브클래스 인스턴스인 start() 메소드를 호출하는 것이다. start() 메소드는 적당한 코드를 갖고 있는 클래스내의 run() 메소드를 실행하며, run() 메소드의 실행에 의하여 새로운 스레드가 생성된다. 반면에 원래의 스레드는 start()를 호출한 다음의 코드를 비동기적으로 계속 실행한다. 또한 스레드는 join() 메소드를 호출하여 다른 스레드가 종료할 때까지 대기할 수 있다[12,14,15].

2.2 데이터 종속성

일반적으로 두 문장 S_1과 S_2 사이의 데이터 종속성은 $S_1 \delta S_2$ 형태로 표현하며 문장 S_2는 S_1에 종속된다고 한다. 데이터 종속성은 (그림 2)와 같이 세 가지 유형으로 분류하여 사용한다. 첫 번째 흐름 종속성(δ^t)은 문장 S_i에서 변수 X가 정의(define)되고 문장 S_j에서 X가 사용(use)되며 S_i가 S_j 이전에 수행된다. 두 번째 출력 종속성(δ^o)은 두 문장 S_i와 S_j 사이에서 같은 변수가 정의되고 S_i가 S_j 이전에 수행된다. 세 번째 역 종속성(δ^a)은 문장 S_i에서 변수 X가 사용하고 문장 S_j에서 X가 정의되며 S_i가 S_j 이전에 수행된다. 또한 데이터 종속성은 세 가지 유형 중 하나만 만족하면 종속이 존재한다고 한다[2,3].



(그림 2) 데이터 종속 종류
(Fig. 2) Type of Data Dependence

또 다른 방식으로는 루프 독립 종속(loop independent dependence, LID)과 루프 연관 종속(loop carried dependence, LCD)으로 구분한 형태가 있다. LID는 루프내의 서로 다른 반복주기들 사이에서는 종속성이 존재하지 않고, 동일한 반복주기내에서만 종속성이 존재하는 경우이다. 즉 같은 반복주기내에서 문장 S_i에서 변수 X가 정의되고 문장 S_j에서 X가 사용되는 경우에 S_j는 S_i에 대하여 LID가 존재한다. LCD는 루프내의 서로 다른 반복주기들 사이에 종속성이 존재하는 경우이다. 즉 루프의 한 반복주기에서 문장 S_i에서 변수 X가 정의되고, 다른 반복주기에서 문장 S_j에서 X가 사용되는 경우에 S_j는 S_i에 대하여 LCD가 존재한다[7].

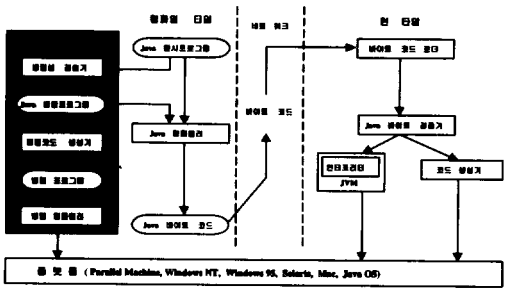
따라서 루프의 형태는 모든 반복주기들 사이에 종속성이 없는 경우와 다른 반복주기 사이에 종속성이 발생하는 경우가 있다. 루프를 펼쳤을 때(unrolled) 모든 문장 각각을 인스

턴스(instance)라 하며 특히, 종속의 근원이 되는 인스턴스를 source 인스턴스, 종속이 되는 인스턴스를 sink 인스턴스라 한다.

DOALL 형태의 루프에서는 반복주기 사이에 종속관계가 존재하지 않기 때문에 한 반복주기내의 데이터들을 프로세서간의 상호 작용없이 병렬처리할 수 있으나 DOACROSS 루프에서는 한 반복주기에서 가져온 데이터가 다른 반복주기에서 수정되거나, 한 반복주기에서 생성된 결과가 나중에 다른 반복주기에서 사용되는 종속관계가 존재하기 때문에 부분적으로 프로세서간의 정보 교환이 필요하게 된다. 이러한 프로세서간의 상호 정보 교환을 동기화 기법이라 하며 프로그램의 올바른 수행을 위해서 반드시 유지되어야 한다.

3. 루프 구조의 병렬성 검출

본 논문은 (그림 3)과 같이 재구성 컴파일러에 의하여 Java 원시 프로그램을 Java의 다중스레드 기법으로 변환하여 수행하는 알고리즘을 제안한다. 따라서 재구성 컴파일러는 Java 프로그램에서 묵시적인 병렬성을 포함하고 있는 여러 개의 단일 루프 구조를 Java의 다중스레드로 변환하여 병렬성을 검출한다. 또한 재구성 컴파일러는 병렬 코드 생성기에 의하여 Java 원시 프로그램을 병렬 코드로 변환하여 병렬 시스템에서 수행할 수 있도록 한다. 재구성 컴파일러는 Java 프로그램에서 묵시적인 병렬성을 다중스레드 의미를 갖는 병렬 루프 구문 DOALL과 DOACROSS 형태로 변환한다.

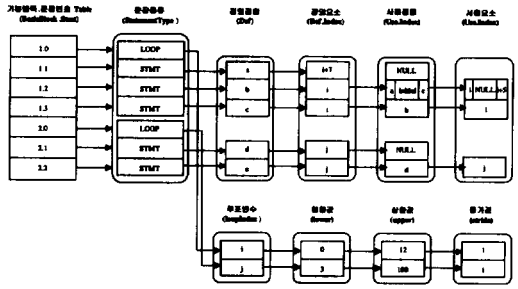


(그림 3) Java의 컴파일과 수행과정
(Fig. 3) Compile-time and run-time of Java

3.1 루프 구조의 심볼 테이블

Java 프로그램의 루프 구조에서 발생하는 모든 심볼은 심볼 테이블에 저장한다. 심볼 테이블의 구성은 블록의 시작과 끝을 제외하고는 블록의 내부 또는 외부로의 분기가 발생하지 않는 단일 루프 구조에서 시작부터 끝까지 순서적으로만 수행하는 문장들의 단위인 기본 블록(basic block)으로 구성한다[1,9]. 본 논문에서는 다음과 같은 SingleThread 클래스가 하나의 스레드를 생성하여 실행하는 Java 프로그램이 존재한다고 가정하며, 이 프로그램을 기본 블록 단위의 심볼 테이블을 구성하면 (그림 4)와 같다.

```
class SingleThread {
    public static void main(String args[]) {
        double[] a,b,c = new double[12];
        double[] d,e = new double[100];
        double initial = 6;
/*L1*/    for (int i=0; i < 12; i++) {
/*S1*/      a[i+7] = 5;
/*S2*/      b[i] = a[i] + initial + c[i+5];
/*S3*/      c[i] = b[i] + 1;
        }
/*L2*/  for (int j=3; j < 100; j++) {
/*S1*/      d[j] = (j + 1) / 2;
/*S2*/      e[j] = d[j] * 3;
        }
    }
}
```



(그림 4) 기본 블록 단위의 심볼 테이블
(Fig. 4) Symbol table of Basic Block

3.2 데이터 종속성 검사

본 논문은 2.2절에서 설명한 세 가지 유형의 데이터 종속성을 Java 프로그램의 루프 구조에 적용하여 검사하였다.

(알고리즘 1)은 기본 블록을 입력으로 받아서 종속이 존재하는 문장 번호와 종속 간격, 종속 종류(흐름 종속, 출력 종속, 역 종속)를 출력하는 종속성 검사 알고리즘이다. sInfor는 기본 블록내의 문장에 대한 문장 번호, 정의 집합, 정의 집합의 첨자, 사용 집합, 사용 집합의 첨자를 의미한다. pInfor는 sInfor 문장을 제외한 기본 블록내의 문장에 대한 문장 번호, 정의 집합, 정의 집합의 첨자, 사용 집합, 사용 집합의 첨자를 의미한다. dElement와 uElement는 각각의 정의 집합과 사용 집합을 의미한다. 예를 들어 (그림 4)에서 sInfor, pInfor, dElement, uElement가 다음과 같을 때,

sInfor = {{1,1}, {a} {i+7}, {NULL}, {NULL}}

pInfor = {{1.2}, {b}, {i}, {a, initial, c},
{i, NULL, i+5}
dElement = {a}
uElement = {a, initial, c}
dElement와 uElement의 교집합은 {a}이므로 종속이 존재하는 문장 번호(1.2), 종속 간격(7), 종속 종류(δ^t)를 저장한다. 이와 같은 방법으로 (알고리즘 1)에 의해 위의 Java 예제 프로그램을 다중스레드 프로그램으로 변환하기 위한 데이터 종속성을 검사하면 (표 1)과 같은 결과를 얻는다.

(알고리즘 1) 데이터 종속성 검사
(Algorithm 1) Data Dependence Test

```
Input : SET BasicBlock
Output : RelatedStmtNo, Distance, Dependence

procedure DataDependenceTest(SET BasicBlock)
 /* 1. True Dependence Test */
        for every sInfor ∈ BasicBlock begin
            for every pInfor exception sInfor ∈ BasicBlock begin
                for every dElement ∈ sInfo.Def begin
                   for every uElement ∈ pInfo.Use begin
                        if (interSection(dElement, uElement)) begin
                            betweenBlockDependency[sInfor.Block][pInfor.RelatedStmtNo.Block] = TRUE
                             sInfor.RelatedStmtNo = ∪ {pInfor.BasicBlock.Stmt}
                             sInfor.Distance = ∪ {dElement.index - uElement.index}
                         sInfor.Dependence = ∪ {δ^t}
                        endif
                    endfor
                endfor
            endfor
        endfor
 /* 2. Output Dependence Test */
     /* 흐름 종속 분석과 동일한 방법으로 두 정의 집합 사이에 종속성을 검사한다. */
 /* 3. Anti Dependence Test */
     /* 흐름 종속 분석과 동일한 방법으로 정의 집합과 사용 집합 사이에 종속성을 검사한다. */
end /* end DataDependenceTest */
 /* Intersection of two set */
procedure interSection(SET α, SET β)
        if ({α ∩ β} ≠ φ ) return TRUE
        else FALSE
end /* end interSection procedure */
```

〈표 1〉 데이터 종속성 검사후의 문장 정보
〈Table 1〉 Statement Information after Data Dependence

기본블럭.문장번호 (BasicBlock.Stmt)	종속이 존재하는 문장번호 (RelatedStmtNo)	종속 간격 (Distance)	종속 종류 (Dependence)	동기화 첨자 (sendWaitIndex)	병렬루프 변환 (Parallel Loop)
1.0					DOACROSS
1.1	{1.2}	{7}	{δ^t}	{0}	
1.2	{1.3, 1.1, NULL, 1.3}	{0, -7, NULL, 5}	{δ^t, δ^t, NULL, δ^a}	{1}	
1.3	{1.2, 1.2}	{-5, 0}	{δ^a, δ^t}		
2.0					DOALL
2.1	{2.2}	{0}	{δ^t}		
2.2	{NULL, 2.1}	{NULL, 0}	{NULL, δ^t}		

3.3 병렬 코드 생성

만약 루프의 모든 반복주기가 독립적이다면, 즉 데이터 종속이 루프에 연관되지 않는 LID 종속이 존재하거나 기본 블록내의 문장간에 전혀 서로 다른 변수를 사용할 때 이 루프는 DOALL 루프로 변환할 수 있다. 비록 데이터 종속이 루프에 연관되는 LCD일지라도 동기화 문장을 삽입하여 DOACROSS 루프로 변환한 후 반복주기를 부분적으로 실행하여 병렬성을 얻을 수 있다[2,3,13].

(표 1)의 데이터 종속성 검사후의 문장 정보 테이블을 이용하여 병렬 구문으로 변환하는 방법은 (알고리즘 2)와 같다.

(알고리즘 2)는 기본 블록을 입력으로 받아서 만약 서로 다른 문장 사이에 종속성이 존재하면 동기화 문장 SEND와 WAIT를 삽입하여 DOACROSS 문장으로 변환하고, 그렇지 않으면 DOALL 문장으로 변환하는 병렬 코드 생성 알고리즘이다. scount는 (표 1)에서 총 종속 간격(Distance) 원소의 개수인 10을 의미하고, sDistance는 종속 간격의 각 원소를 의미한다. 만약 sDistance가 0보다 크다면 종속이 존재하는 문장 아래에 SEND 문장과 동기화 첨자(sendWaitIndex)의 값을 삽입하고, 그렇지 않으면 종속이 존재하는 문장 위에 WAIT 문장, 동기화 첨자의 값, 종속 간격을 삽입한다. 그리고 만약 하나의 루프 구조내에 동기화 문장이 존재하면 반복 루프문 for를 DOACROSS로 변환하고, 그렇지 않으면 반복 루프문 for를 DOALL 문장으로 변환한다.

따라서, (알고리즘 2)에 의해 위의 Java 예제 프로그램의 루프 구조를 병렬 코드로 변환하면 다음과 같다.

```
/*L1*/  DOACROSS (int i=0; i < 12; i++) {
/*S1*/  a[i+7] = 5;
            SEND(0);
```

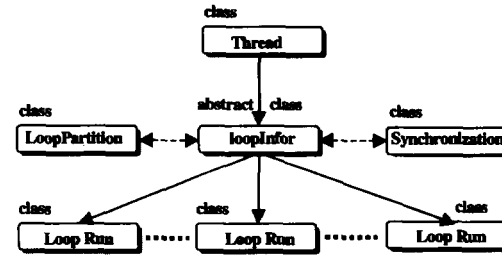
(알고리즘 2) 병렬 코드 생성
(Algorithm 2) Parallel Code Generation

```
Input : SET BasicBlock
Output : Parallel Code

procedure GenerateParallelCode(SET BasicBlock)
        scount = DistanceElementcount(BasicBlock)
        sendWaitIndex = 0
        for every sInfor ∈ BasicBlock begin
            generate(source(SYMBOL(sInfor.BasicBlock.Stmt)));
            for every sDistance ∈ sInfor.Distance begin
                    if (sDistance ≡ NULL || sDistance ≡ {0} ) count++
                    else if (sDistance > 0) then
                      generate(sInfor.BasicBlock.Stmt+1, "SEND("+sendWaitIndex+")");
                      sInfor.sendWaitIndex=sendWaitIndex;
                      sendWaitIndex++;
                    else if (reference(sInfor.RelatedStmtNo).sendWaitIndex ≠ NULL) then
                      generate(sInfor.BasicBlock.Stmt-1,"Wait("+reference(sInfor.RelatedStmtNo).
                                  sendWaitIndex+","+"loopIndex(sInfor.Distance)"+"-"+sDistance);
                      sInfor.sendWaitIndex = reference(sInfor.RelatedStmtNo).sendWaitIndex;
                    else
                       generate(sInfor.BasicBlock.Stmt-1, "Wait("+sendWaitIndex+","
                                  +"loopIndex(sInfor.Distance)"+"-"+sDistance);
                      sInfor.sendWaitIndex = sendWaitIndex;
                       sendWaitIndex++;
                    endif
            endfor
                    if (scount ≡ count)  replace(loopPos, "DOALL")
                    else  replace(loopPos, "DOACROSS")
        endfor
end /* end GenerateParallelCode */
```

```
        WAIT(0, i-7);
/*S2*/  b[i] = a[i] + initial + c[i+5];
        SEND(1);
        WAIT(1, i-5);
/*S3*/  c[i] = b[i] + 1;
      }
/*L2*/  DOALL (int j=3; j < 100; j++) {
/*S1*/    d[j] = (j + 1) / 2;
/*S2*/    e[j] = d[j] * 3;
      }
```

3.4 루프 구조의 다중스레드로 변환

본 논문은 Java 언어에서 병렬 루프를 구현하기 위하여 (그림 4)와 같이 클래스 계층구조를 갖는다. Java에서 스레드는 Java API의 java.lang 패키지의 Thread 클래스를 사용하여 생성한다[11,15]. 추상화 클래스 loopInfor는 Thread 클래스를 상속받고 루프 구조에 필요한 하한값과 상한값을 갖는다. LoopRun 클래스는 loopInfor를 상위 클래스로 하는 클래스를 선언하고, run() 함수를 생성하여 원시 프로그램의 루프를 복사한다. LoopPartition 클래스는 블록 스케줄링 방법을 이용하여 모든 반복주기를 프로세서에 할당한다. Synchronization 클래스는 여러 개의 프로세서가 공유 데이터를 접근할 때 동기화를 부여한다.



(그림 4) 클래스 계층 구조
(Fig. 4) Class Hierarchy

(알고리즘 3) 프로세서 할당 클래스
(Algorithm 3) Processor Allocation Class

```
Input : lower-upper, Processor Number, Loop Number
Output : Partition of lower-upper bound

procedure class Partition(lowhigh, ProcessorN, forCount, lbub)
 /* 1. 루프의 하한값과 상한값을 초기화 */
        for every sInfor ∈ BasicBlock begin
            for every sInfor.StatementType = {LOOP} begin
                lowhigh[i][j] = reference(sInfor.Def).lower;
                lowhigh[i+1][j] = reference(sInfor.Def).upper;
                j += 2;
            end
        end
 /* 2. 블록 크기를 결정 */
        for every i = [0..forCount*2] step 2 begin
                temp[i] = lowhigh[0][i];
                blocksize[i] = (integer)ceil((lowhigh[1][i] - lowhigh[0][i]) / ProcessorN);
        end
 /* 3. 프로세서 수에 따라 반복주기를 분할하여 2차원 배열에 할당 */
        for every i = [0..forCount*2] step 2 begin
            for every j = [0..ProcessorN] step 1 begin
                lowhigh[0][i] = temp[i];
                lbub[j][i] = lowhigh[0][i];
                lbub[j][i+1] = min(lbub[j][i] + blocksize[i], lowhigh[1][i]);
                temp[i] = lbub[j][i+1];
            end
        end
end /* end of Partition procedure class */
```

다중스레드 변환 알고리즘은 프로세서 할당 알고리즘, 동기화 문장 삽입 알고리즘, 주 프로그램 변환 알고리즘의 세부분으로 구성된다.

3.4.1 프로세서 할당

공유메모리 구조에서 병렬 루프는 루프의 서로 다른 반복주기를 다중 프로세서에 할당하여 병렬로 실행한다. Java에서는 루프의 서로 다른 반복주기를 다중스레드에 할당하여 fork-join 형태의 병렬성을 이용하여 실행할 수 있다. 스레드에 할당된 반복주기는 스케줄링 방법에 종속한다. 본 논문에서는 모든 반복주기가 똑같은 작업을 수행하는 정적 할당 블록 스케줄링 방법을 이용한다[9].

(알고리즘 3)은 각 루프의 하한값과 상한값, 프로세서 수, 반복 구문 for 문의 수를 입력으로 받아서 프로세서 수에 따른 2차원 배열에 분할하는 프로세서 할당 알고리즘이다. 알고리즘의 수행 과정은 첫 번째 각 루프의 하한값과 상한값을 2차원 배열 lowhigh 변수에 저장한다. 두 번째 각 프로세서에 정적 할당되는 블록 크기(blocksize)를 결정하며, 블록 크기는 다음의 수식에 의하여 결정된다. temp 변수는 블록 할당을 위한 임시 변수이다.

$$blocksize = \lceil \frac{upper - lower}{processor\ No} \rceil$$

세 번째 각 루프 문장별로 프로세서 수에 따른 하한값과 상한값을 일정하게 분할하여 2차원 배열 lbub 변수에 저장한다. (알고리즘 3)에 의해 위의 Java 예제 프로그램을 4개의 프로세서를 가진 병렬 시스템에 할당한다고 가정할 경우에 할당된 결과는 (표 2)와 같다.

〈표 2〉 4개의 프로세서에 루프 분할
〈Table 2〉 Loop partition with four processor

for-stmt	bound	processor 1	processor 2	processor 3	processor 4
1st	lower	0	3	6	9
	upper	3	6	9	12
2nd	lower	3	28	53	78
	upper	28	53	78	100

3.4.2 동기화 방법

Java 언어에서 하나의 객체를 두 개 이상의 스레드가 공유하여 수행한다면 일관된 데이터의 값을 유지할 수 없게될 가능성이 있다. 따라서 한 번에 하나의 스레드만 객체를 접근하도록 제어할 필요가 있는데 이것을 동기화라 하며, Java에서는 동기화를 위해 모니터의 잠금(lock)을 사용한다. 동기화는 두 개의 스레드를 상호배제로 수행하게 된다[14,15].

다중스레드 환경에서 사용 가능한 클래스를 만들기 위해서 메소드들은 synchronized로 선언되어야 한다. 스레드가 하나의 객체상에서 synchronized로 선언된 메소드를 호출하면, 그 객체에 잠금이 설정되어 다른 스레드의 사용을 막는다. 따라서 동일한 객체를 접근하는 다른 스레드가 동기화된 메소드를 호출한다 해도 이미 객체에 설정된 잠금이 해제될 때까지 블록된다.

(알고리즘 4)는 DOACROSS 루프에서 동기화를 요구하는 변수를 입력으로 받아서 동기화 변수의 특정 비트를 잠금하는 send 메소드와 동기화 변수의 특정 비트가 잠금될 때까지 블록킹하는 wait 메소드를 사용한다.

(알고리즘 4) 동기화 클래스
(Algorithm 4) Synchronization Class

```
Input : Synchronization Variable
Output : Lock or Unlock

procedure class Sync(syncAlloc)
      lock = new boolean[SyncAlloc][SyncAlloc]
 /* 1. Send Method */
      synchonized send(i, j) begin
           lock[i][j] = TRUE;
           notifyAll();
      end
 /* 2. Wait Method */
      synchonized wait(i, j) begin
           while (!lock[i][j])
                wait();
      end
end /* end Synchronization procedure class */
```

boolean 배열 lock은 TRUE와 FALSE 중 하나의 값을 갖는 비트 배열을 사용하고, 동기화 변수의 반복주기 i 비트를 잠금하는 것은 동기화 send 메소드를 불러서 실행한다. send 메소드에서 notifyAll()을 호출하는 것은 어떤 동기화 변수에 의하여 블록된 스레드를 깨우는 것이다. 반면에 동기화 wait 메소드를 호출하는 것은 반복주기 j 비트가 잠금될 때까지 동기화 변수는 블록한다. 이 구현에서 병렬 DOACROSS 루프가 프로세서 사이에 자주 접근하는 동기화 변수로 인하여 wait 메소드를 호출할 때는 오버헤드를 초래하지만 일

부분의 병렬성을 얻을 수 있다.

3.4.3 주 프로그램 변환

주 프로그램 변환은 main 클래스 생성, 원시 프로그램의 루프 변환, 루프 구조의 추상화 클래스 생성으로 구성한다.

첫 번째 main 클래스 생성은 (알고리즘 5)와 같으며, 다음과 같은 문장을 생성한다.

· 프로세서 수에 대한 변수 선언과 calcLohi 함수

(알고리즘 5) 주 프로그램 클래스
(Algorithm 5) Main Program Class

```
Input : Dependence test information
Output : Main program code generation

procedure class MainGenerate()
        generate(ProcessorN, forCount = calcLoHi)
        generate(lowhigh = new [2][forCount * 2])
        generate(lbub = new [ProcessorN][forCount * 2])
        generate(declarlation(SYMBOL)) /* 심볼 테이블로부터 원시 프로그램의 선언문 생성 */
        generate(call Partition(lowhigh, ProcessorN, forCount, lbub)) /* 스케쥴링 부분 호출 문장 삽입 */
        generate(Sync P = new Sync()) /* 문장 동기화 메모리 할당 */
        generate(call Sync(syncAlloc))
        /* 루프 구조 생성 및 스레드 생성 호출 */
        if (forCount ≡ 1) begin  /* 단일 for문장일 경우 */
                generate(call genLoop(lbub[0], basicBlock[0], P, i+1)
                generate(for j = [0..ProcessorN])
                generate(call loop+(0)(lbub[0], declaration(BasicBlock[0]), P)
                generate(start)
        end
        for (i = 1 ; i < forCount; i++) begin   /* 2개 이상의 for 문장일 경우 */
                if (betweenBlockDependence[i][i+1] ≠ TRUE) begin     /* 두 BasicBlock 사이에 병합 */
                        BasicBlock = BasicBlock ∪ {{BasicBlock[i]} ∪ {BasicBlock[i+1]}}
                        lhMrg = lhMrg ∪ {{lbub[i]} ∪ {lbub[i+1]}}
                end
                else begin  /* BasicBlock간에 종속이 존재할 경우 */
                        generate(call genLoop(lhMrg, BasicBlock, P, i)
                        generate(for j = [0..ProcessorN])
                        generate(call loop+i(lhMrg, declaration(BasicBlock), P)
                        generate(start)
                        generate(Barrier());
                        generate(call genLoop(lbub[i+1], basicBlock[i+1], P, i+1)
                        generate(for j = [0..ProcessorN])
                        generate(call loop+(i+1)(lbub[i+1], declaration(BasicBlock[i+1]), P)
                        generate(start)
                        i++;
                        BasicBlock = {ϕ}
                        lhMrg = {ϕ}
                end;
        end;
        if (BasicBlock ≠ {ϕ}) begin  /* BasicBlock간에 종속이 없을 경우 */
                generate(call genLoop(lhMrg, BasicBlock, P, i)
                generate(for j = [0..ProcessorN])
                generate(call loop+i(lhMrg, declaration(BasicBlock), P)
                generate(start)
        end
end /* end MainGenerate procedure class */

procedure calcLoHi()   /* for 문장 개수와 동기화 변수인 비트 배열을 할당하기 위한 크기를 얻는다. */
        for every sInfor ∈ BasicBlock begin
                for every sInfor.StatementType ≡ {LOOP} begin
                        forCount++
                        generate(reference(sInfor.Def).lower, reference(sInfor.Def).upper)
                        syncAlloc= max(syncAlloc, reference(sInfor.Def).upper)
                endfor
        endfor
        return forCount
end /* end procedure calcLoHi */
```

호출을 통해 for 문장 개수 선언
- 프로세서 할당에 필요한 배열의 하한값과 상한값을 저장하기 위한 lbub 변수를 선언
- 심볼 테이블로부터 원시 프로그램의 선언문을 삽입
- 스케줄링 호출 문장 삽입
- 동기화 변수인 비트 배열의 메모리를 할당하기 위한 문장 호출
- 루프 구조 생성 및 스레드 생성 호출

루프 구조 생성 및 스레드 생성 호출 부분은 프로그램 내에 for 문장의 개수가 하나일때와 n개 일때로 나누어 처리한다. for 문장의 개수가 하나일때는 프로세서 개수만큼의 스레드를 생성하여 실행한다. for 문장의 개수가 하나 이상일 때 인접한 for 루프 L_i와 L_j 사이에 종속이 존재하지 않는다면 두 개의 루프는 하나의 클래스내에서 실행할 수 있다는 것을 의미하기 때문에 병합한다. 예를 들면 BasicBlock[1]과 BasicBlock[2]가 존속이 존재하지 않는다면 BasicBlock은 {1, 2} 정보를 갖는다. 또한 각 for 문장에 대한 배열의 하한값과 상한값을 병합한다. 즉, 첫 번째 for 문장 lbub[1]의 하한값과 상한값이 {lbub[0][0], lbub[0][1]}이고, 두 번째 for 문장 lbub[2]의 하한값과 상한값이 {lbub[0][2], lbub[0][3]}이라면 병합한 lhMrg는 {lbub[0][0], lbub[0][1], lbub[0][2], lbub[0][3]}가 된다.

그러나 인접한 for 루프 L_i와 L_j 사이에 종속이 존재한다면 루프 L_i와 L_j는 서로 다른 클래스에서 실행하기 위한 루프 구조 생성과 스레드 호출 부분을 삽입하고 루프 L_i와 L_j 사이에 Barrier 동기화 문장을 삽입한다. 예를 들면 BasicBlock[1..3] = {1, 2, 3}에 대해 **BasicBlock[1]과 BasicBlock[2]는 종속이 존재하지 않고 BasicBlock[3]은 BasicBlock[2]에 대해 종속이 존재한다면 BasicBlock[1..2] = {1, 2}는 동일한** 클래스에서 실행할 수 있지만 BasicBlock[3] = {3}은 동일한 클래스에서 실행할 수 없으므로 루프를 분할하여 동기화 문장을 삽입해야 한다.

이와 같은 방법으로 (알고리즘 5)에 의해 위의 Java 예제 프로그램의 main 클래스를 생성하면 다음과 같다.

```
class Main {
  static loop2[] t;
  public static void main(String args[]) {
    /* 프로세서 수, for 문장의 수, 루프의 하한값
       과 상한값 초기화 */
    /* 사용한 모든 배열의 메모리 할당 */
    /* 알고리즘 3 호출 */
    for (int i = 0; i < p_no; i++) {
      t[i] = new loop2(lb_ub[i][0], lb_ub[i]
             [1], lb_ub[i][2], lb_ub[i][3],
             a, b, c, d, e, p, initial);
      t[i].start();
    }
  }
}
```

두 번째 원시 프로그램의 루프 변환은 (알고리즘 6)과 같으며, 다음과 같은 문장을 생성한다.
- loopInfor를 상위 클래스로 하는 클래스 loop+i 선언
- 루프내에서 사용되거나 정의된 변수의 선언
- loop+i 메소드의 매개변수로 배열의 하한값과 상한값 리스트, 루프에서 정의되거나 사용된 변수의 선언, 동기화 변수 선언
- 매개변수로 받은 값을 멤버 데이터에 할당
- run() 메소드 생성
- 각각의 루프에 대하여 먼저 생성한 병렬 코드를

(알고리즘 6) 루프 변환 클래스
(Algorithm 6) Loop Replacement Class

```
Input : merge value of lower-upper bound,
        define-use set of loop, synchronization variable
Output : construction of concurrent execution part

procedure class genLoop(lhMrg, LoopMrg, P, i)
  generate(class + className(loop+i) +
           extends loopInfor)
  generate(declaration(LoopMrg))
  generate(public + className(loop+i) + declaration(
           lhMrg, declaration(LoopMrg), Sync P)
  generate(this.declaration(LoopMrg) ←
           declaration(LoopMrg))
  generate(public + void + run())
    for every BasicBlock∈BasicBlock(LoopMrg) begin
      generate(for lhMrg(i)[0]; i < lhMrg(i)[1]; i++)
          source=generate(GenerateParallelCode(
                          CurrentBasicBlock))
    end
    replace(source,[DOALL, DOACROSS], for)
end /* end genLoop class */
```

호출하여 삽입

· 병렬 코드 DOALL과 DOACROSS 문장을 for 문으로 변환

(알고리즘 6)에 의해 위의 Java 예제 프로그램의 루프 변환 클래스를 생성하면 다음과 같다.

```
class loop2 extends loopInfor {
  /* 루프내에서 사용되거나 정의된 변수의 선언 */
   public loop2(int l1, int h1, int l2, int h2,
   double[] a, double[] b, double[] c,
   double[] d, double[] e, Sync p, double
   initial) {
        super(l1, h1, l2, h2, p);
        /* 매개 변수로 받은 값을 멤버 데이터에
           할당 */
    }
    public void run() {
        for(int i = l1; i < h1; i++) {
          /* 동기화 문장이 삽입된 원시 프로그램
             의 첫 번째 루프 부분을 복사 */
        }
        for(int j = l2; j < h2; j++) {
          /* 원시 프로그램의 두 번째 루프 부분
             을 복사 */
        }
    }
}
```

세 번째 루프 구조의 추상화 클래스 생성은 (알고리즘 7)과 같으며, 추상화 클래스 loopInfor는 Thread 클래스를 계승받고 루프 구조에 필요한 하한값과 상한값을 가지고 있는 클래스이다. 이 클래스는 매개변수로 입력받은 배열의 하한값과 상한값을 단지 멤버 데이터에 할당하는 역할을 수행한다.

(알고리즘 7)에 의해 위의 Java 예제 프로그램의 루프 구조의 추상화 클래스를 생성하면 다음과 같다.

```
class loopInfor extends Thread {
      /* run() 메소드에서 사용할 루프의 하한값과
         상한값, 동기화 변수를 선언 */
      public loopInfor(int l1, int h1, int l2,
      int h2, Sync p) {
      /* 매개 변수로 받은 값을 멤버 데이터에 할
         당 */
    }
}
```

(알고리즘 7) 루프 구조의 추상화 클래스
(Algorithm 7) Abstract Class of Loop Structure

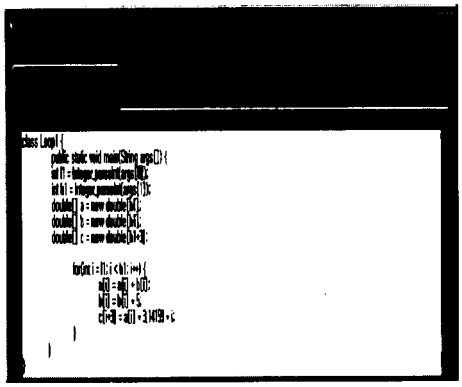
```
Input : Argument list
Output : member data allocation

procedure class genloopInfor(argument_list ...)
    generate(class + loopInfor + extends + Thread)
    generate(declaration(argument_list...)
    generate(public + loopInfor +
              (declaration(argument_list ...)));
    generate(this. + subtract(argument_list, TYPE)
                   = subtract(argument_list, TYPE))
end /* end genloopInfor class */
```

4. 적용 및 분석 결과

본 논문의 프로그래밍 환경은 Sun Solaris 2.5 운영체제를 가지고 있는 Sun Ultra SPARC상에서 JDK1.1.5 Final 버전을 적용하여 분석하였다. 루프의 형태는 반복주기 사이에 종속성이 없을 때 사용하는 DOALL 루프 구조와 다른 반복주기 사이에 종속성이 발생하는 경우에 사용하는 DOACROSS 루프 구조로 나누어 적용하였다.

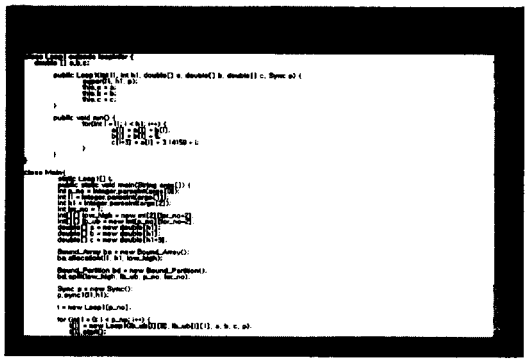
4.1 DOALL 루프 구조

(그림 6)과 같이 하나의 for 문장 구조를 가지고 있고 종속성 분석에 의하여 반복주기 사이에 종속이 존재하지 않는 Loop1 클래스가 있다고 가정하자.



(그림 6) 예제 프로그램-1
(Fig. 6) Example program-1

(그림 6)을 제안한 알고리즘에 의하여 묵시적인 병렬성을 검출하여 Java 스레드 프로그램으로 변환하면 (그림 7)과 같다.



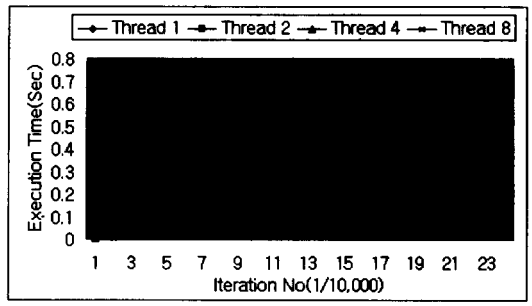
(그림 7) 그림 6의 스레드 프로그램
(Fig. 7) Thread program of figure 6

본 논문에서 적용한 실행 시간은 원시 Java 예제 프로그램은 스레드가 한 개이고, 변환된 다중스레드 Java 프로그램에서는 스레드의 수가 2개, 4개, 8개를 가질 때로 나누어 적용하였다. 따라서 다중스레드의 수는 병렬 시스템의 다중 프로세서에 할당된다고 가정한다.

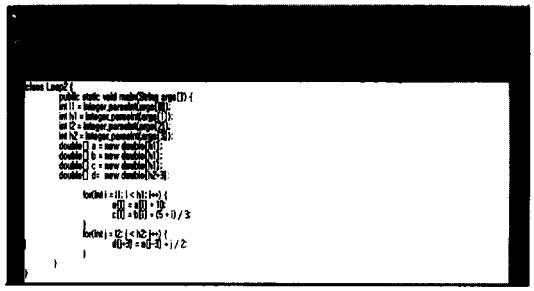
(그림 8)은 (그림 6)과 (그림 7)의 Java 프로그램에서 반복주기 개수 N의 값 변화에 따른 실행 시간을 나타낸다. Loop1 클래스는 동기화 문장이 존재하지 않기 때문에 반복주기의 수가 증가함에 따라 실행 시간이 선형적으로 증가함을 알 수 있다. 원시 프로그램에서 스레드가 1개인 프로그램 보다 변환된 프로그램에서 스레드가 8개일 경우에 반복주기 수가 100,000에서 86%의 성능이 향상되었다. 따라서 본 논문에서 제안한 방법에 따라 Java 원시 프로그램을 다중스레드 프로그램으로 변환하여 실행할 경우 프로세서 수가 증가함에 따라 실행 시간이 선형적으로 단축됨을 알 수 있다.

(그림 9)와 같이 2개의 for 문장 구조를 가지고 있고 종속성 분석에 의하여 반복주기 사이에 종속이 존재하지 않는 Loop2 클래스가 있다고 가정하자.

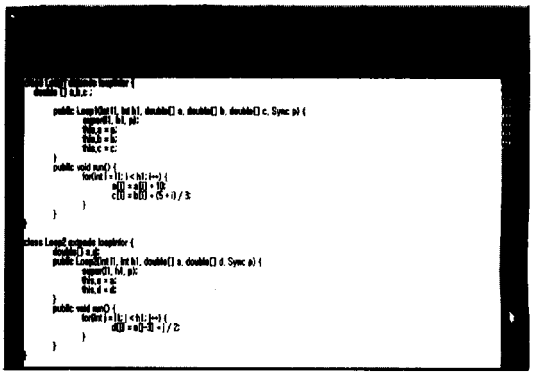
(그림 9)를 제안한 알고리즘에 의하여 묵시적인 병렬성을 검출하여 Java 스레드 프로그램으로 변환하면 (그림 10)과 같으며, 2개의 for 루프 사이에 종속성이 존재하므로 Main 클래스에서 Barrier 동기화 문장을 삽입하고 2개의 Loop1과 Loop2 클래스에서 실행할 수 있다.



(그림 8) Loop1 클래스의 성능 분석
(Fig. 8) Performance analysis of Loop1 class



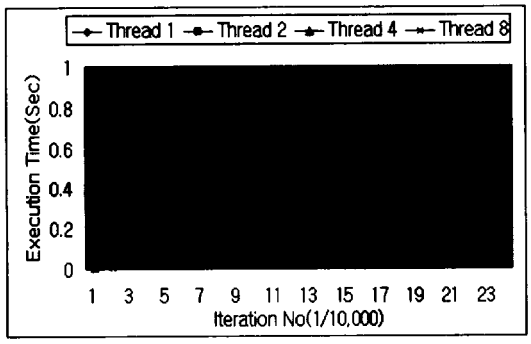
(그림 9) 예제 프로그램-2
(Fig. 9) Example program-2



(그림 10) 그림 9의 스레드 프로그램
(Fig. 10) Thread program of figure 9

(그림 11)은 (그림 9)와 (그림 10)의 Java 프로그램에서 반복주기 개수 N의 값 변화에 따른 실행 시간을 나타낸다. (그림 10)은 (그림 7)과 같이 반복주기 사이에 동기화 문장이 존재하지 않기 때문에 반복주기의 수가 증가함에 따라 실행 시간이 선형적으로 증가함

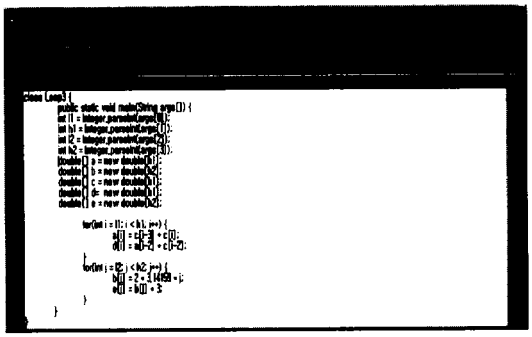
을 알 수 있다. 따라서 본 논문에서 제안한 방법에 따라 Java 원시 프로그램을 다중스레드 프로그램으로 변환하여 실행할 경우 프로세서 수가 증가함에 따라 실행 시간이 선형적으로 단축되며, 두 개의 루프 사이에 Barrier 동기화 문장으로 인한 약간의 오버헤드가 있음을 알 수 있다.



(그림 11) Loop2 클래스의 성능 분석
(Fig. 11) Performance analysis of Loop2 class

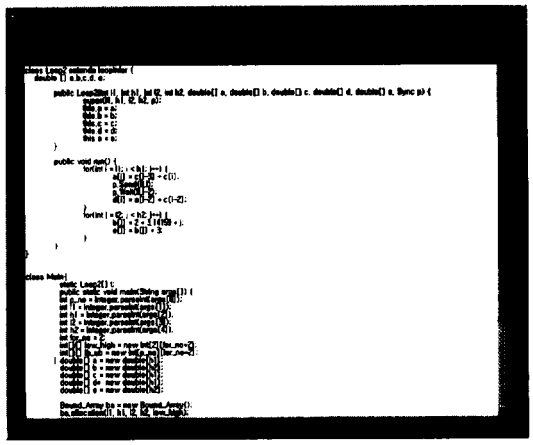
4.2 DOACROSS 루프 구조

(그림 12)와 같이 2개의 for 문장 구조를 가지고 있고 종속성 분석에 의하여 반복주기 사이에 종속이 존재하는 Loop3 클래스가 있다고 가정하자.



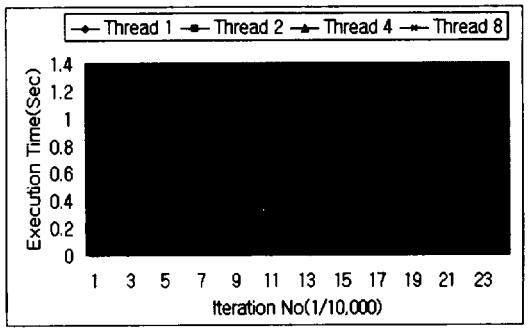
(그림 12) 예제 프로그램-3
(Fig. 12) Example program-3

(그림 12)를 제안한 알고리즘에 의하여 묵시적인 병렬성을 검출하여 Java 스레드 프로그램으로 변환하면 (그림 13)과 같으며, 첫 번째 for 루프는 반복주기 사이에 종속성이 존재하므로 동기화 문장 send와 wait를 삽입하였고, 2개의 for 루프 사이에는 종속성이 존재하지 않으므로 하나의 클래스에서 실행할 수 있다.



(그림 13) 그림 12의 스레드 프로그램
(Fig. 13) Thread program of figure 12

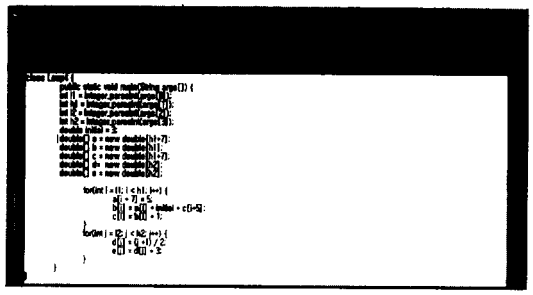
(그림 14)는 (그림 12)와 (그림 13)의 Java 프로그램에서 반복주기 개수 N의 값 변화에 따른 실행 시간을 나타낸다. (그림 14)는 (그림 13)의 변환된 다중스레드 프로그램이 1개의 동기화 문장을 포함하므로 스레드가 2개일 때는 (그림 12)의 단일 스레드 프로그램보다 실행 시간이 느리지만 스레드가 4개 이상일 때는 실행 시간이 향상됨을 알 수 있다. 원시 스레드가 1개인 프로그램 보다 변환된 프로그램에서 스레드가 8개일 경우에 반복주기 수가 100,000에서 71%의 성능이 향상되었다.



(그림 14) Loop3 클래스의 성능 분석
(Fig. 14) Performance analysis of Loop3 class

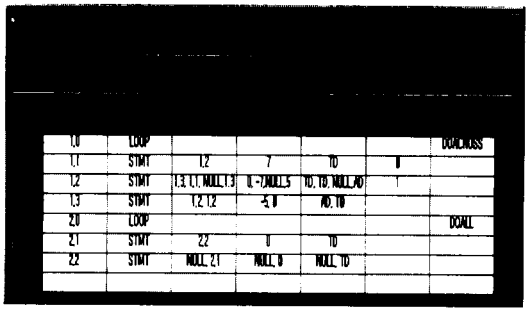
(그림 15)와 같이 2개의 for 문장 구조를 가지고 있

고 종속성 분석에 의하여 반복주기 사이에 종속이 존재하는 Loop4 클래스가 있다고 가정하자.



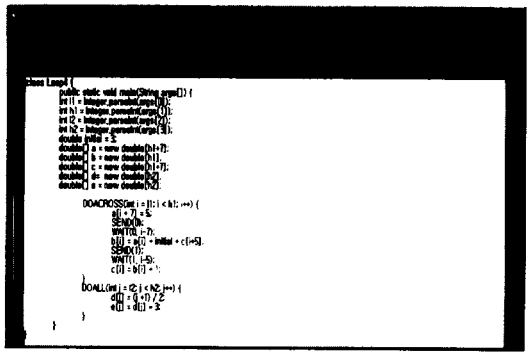
(그림 15) 예제 프로그램-4
(Fig. 15) Example program-4

(그림 15)를 제안한 종속성 분석 알고리즘에 의하여 종속성을 분석하면 (그림 16)과 같다.



1.0	LOOP					DOACROSS
1.1	STMT	1.2	7	TO	8	
1.2	STMT	1.3 1.1, NULL 1.3	0 -7,NULL 5	TO, TO, NULL,A0	1	
1.3	STMT	1.2 1.2	-5 8	A0, TO		
2.0	LOOP					DOALL
2.1	STMT	2.2	8	TO		
2.2	STMT	NULL 2.1	NULL 8	NULL TO		

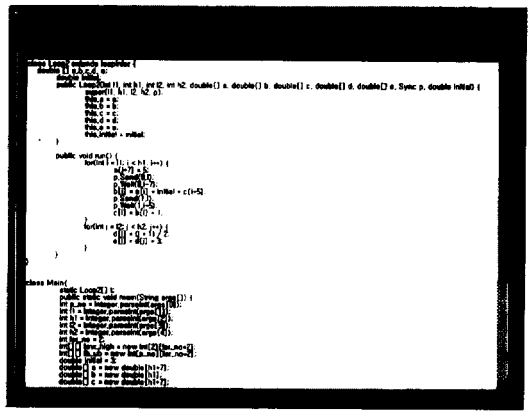
(그림 16) Loop4 클래스의 종속성 분석
(Fig. 16) Data dependence information of Loop4 class



(그림 17) Loop4 클래스의 병렬 프로그램
(Fig. 17) Parallel program of Loop4 class

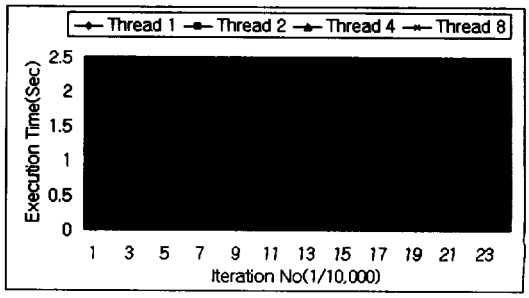
(그림 16)의 데이터 종속성 분석 정보를 가지고 제안한 병렬 프로그램 변환 알고리즘에 의하여 병렬 프로그램으로 변환하면 (그림 17)과 같다.

(그림 15)를 제안한 알고리즘에 의하여 묵시적인 병렬성을 검출하여 Java 스레드 프로그램으로 변환하면 (그림 18)과 같으며, 첫 번째 for 루프는 반복주기 사이에 종속성이 존재하므로 동기화 문장 send와 wait를 삽입하였고, 2개의 for 루프 사이에는 종속성이 존재하지 않으므로 하나의 클래스에서 실행할 수 있다.



(그림 18) 그림 15의 스레드 프로그램
(Fig. 18) Thread program of figure 15

(그림 19)는 (그림 15)의 원시 프로그램과 (그림 18)의 변환된 Java 다중스레드 프로그램에서 반복주기 개수 N의 값 변화에 따른 실행 시간을 나타낸다. (그림 19)에서 다중스레드 프로그램은 2개의 동기화 문장을 포함하므로 스레드가 2개일 때는 단일 스레드 프로그램보다 성능이 저하되고, 스레드가 4개일 때는 약간 좋아지지만 반복주기 수가 100,000에서 스레드가 8개일 때는 57% 정도의 성능이 향상됨을 알 수 있다.



(그림 19) Loop4 클래스의 성능 분석
(Fig. 19) Performance analysis of Loop4 class

5. 결 론

순차 Java 프로그램을 병렬 프로그램으로 변환하는 방식에는 사용자에 의한 방법과 재구성 컴파일러에 의한 방법이 있다. 사용자에 의한 방법은 사용자가 새로운 병렬 프로그래밍 언어를 습득해야 하는 오버헤드가 있고 병렬성 검출을 사용자에게 요구하는 단점이 있다. 재구성 컴파일러에 의한 방법은 프로그래머가 병렬성을 명시하거나 찾아낼 필요가 없다는 장점이 있다.

따라서 기존의 순차적 Java 프로그램을 병렬 시스템에서 재사용하기 위해서는 Java 언어 자체에서 지원하는 다중스레드 프로그램으로 변환하는 것이 요구된다. 일반적으로 응용 프로그램에서 루프 구조는 전체 수행 시간 중 많은 부분을 차지한다.

본 논문은 기존에 작성된 Java 프로그래밍 언어에서 단일 루프 구조에서 데이터 종속성 분석을 수행한 후 묵시적 병렬성을 검출하여 병렬 코드를 생성하였다. 또한 Java 원시 프로그램을 언어의 다중스레드 기법으로 변환하고, 재구성 컴파일러에 의하여 병렬성을 검출하는 방법을 제안하였다. 그리고 다중스레드 문장으로 변환된 프로그램에 대해 루프의 반복계수와 스레드 수를 매개변수로 하여 여러 측면에서 비교·분석하였다. 분석 결과에 의하면 본 논문에서 제안한 알고리즘에 의해 다중스레드로 변환한 프로그램은 반복주기 수가 증가함에 따라 원시 Java 프로그램 보다 실행 시간이 50% 이상 단축됨을 알 수 있었다.

본 논문에서 제안한 재구성 컴파일러에 의한 이점은 사용자의 병렬성 검출에 대한 오버헤드를 줄이고, 순차 Java 프로그램에 대한 효과적인 병렬성 검출을 가능하게 한다. 앞으로는 다중 중첩 루프 구조에 대한 묵시적 병렬성을 검출하는 연구가 필요하다.

참 고 문 헌

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman, "Compilers Principles, Techniques and Tools," Addison-Wesley, 1986.

[2] Michael J. Wolfe. "Optimizing Supercompilers for Supercomputers," MIT Press, 1989.

[3] Hans Zima and Barbara Chapman. "Supercompiler for Parallel and Vector Computers," ACM Press, 1991.

[4] Ken Arnold and James Gosling, "The Java Programming Language," Addison-Wesley, Reading, Massachusetts, 1996.

[5] Won Kim, Frederick H. Lochovsky, "Object Oriented Concepts, Databases, and Applications," Addison Wesley, pp.79-124, 487-520, 1989.

[6] Doug Lea, "Concurrent Programming in Java," Addison-Wesley, Reading, Massachusetts, 1997.

[7] Utpal Banerjee, "Dependence Analysis for Supercomputing", Kluwer, Boston, 1988.

[8] Scott Oaks and Henry wong, "Java Threads," O'Reily & Associates, Sebastopol, CA, 1997.

[9] Constantine D. Polychronopoulos. "Parallel Programming and Compilers," Kluwer, Boston, 1988.

[10] Radenski, A. A, "Object-Oriented Programming and Parallelism : Introduction," Inf. Sci(USA), Vol.93, No.1-2, pp.1-7, August. 1996.

[11] Michael Morrison, "Java 1.1 Third Edition," Sams.net, 1997.

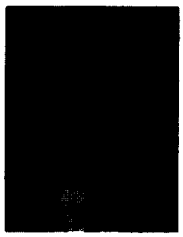
[12] H. M. Deitel and P. J. Deitel, "Java, How to Program," Prentice-Hall, 1997.

[13] Cheng-Tien Wu, Chao-Tung Yang, Shian-Shyong Tseng, "PPD: A practical parallel loop detector for parallelizing compilers," Proceedings. 1996 International Conference on Parallel and Distributed Systems, pp. 274-281, 1996.

[14] Tim Lindholm and Frank Yellin, "The Java Virtual Machine Specification," Addison-Wesley, Reading, Massachusetts, 1996.

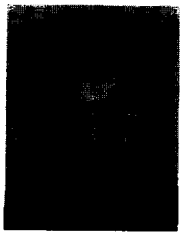
[15] James Gosling, Bill Joy, and Guy Steele, "The Java Language Specification," Addison-Wesley, Reading, Massachusetts, 1996.

[16] 황득영, 정계동, 최영근, "병렬 프로그램의 사건기반 디버깅에 의한 효율적인 데이터 레이스 검출," 병렬처리시스템학술발표회논문집, 제3권, 제2호, pp.54-63, 1992.



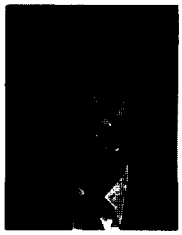
황 득 영

1988년 광운대학교 전자계산학과 (이학사)
1990년 광운대학교 전자계산학과 (이학석사)
1990년~1994년 기전여자 전문대학 조교수
1991년~현재 광운대학교 전자계산학과 박사과정
1994년~현재 삼척산업대학교 컴퓨터과학과 조교수
관심분야 : 병렬 컴파일러, 병렬 프로그래밍 언어, 객체지향 프로그래밍 언어, 시각언어, 분산처리



권 오 진

1990년 광운대학교 전자계산학과 (이학사)
1990년~1992년 수도군단 유선소대장
1994년 광운대학교 대학원 전자계산학과(이학석사)
1994년~현재 산업기술정보원 데이터베이스 사업부 연구원
관심분야 : 병렬 컴파일러, 객체지향 소프트웨어, 검색지향 소프트웨어, 검색 시스템, 시각언어



최 영 근

1980년 서울대학교 사범대학교 수학교육과(이학사)
1982년 서울대학교 계산통계학과 (이학석사)
1989년 서울대학교 계산통계학과 (이학박사)
1983년~현재 광운대학교 전자계산학과 교수
1997년~현재 광운대학교 전자계산소 소장
관심분야 : 병렬 컴파일러, 병렬 프로그래밍 언어, 객체지향 프로그래밍 언어, 객체지향 분산 컴퓨팅