자바 프로그래밍에서 병렬처리를 위한 중첩 루프 구조의 다중스레드 변환

황 득 영† · 최 영 근††

요 약

병렬 시스템에서 순차 자바 프로그램을 재 사용할 수 있기 위해서는 자바 프로그램 내에 존재하는 병렬성을 찾아내는 것이 중요하다. 자바 프로그램을 병렬 시스템에서 실행할 경우 루프는 전체 수행 시간 중 많은 부분을 차지하므로 병렬성 검출의 기본이 되지만 데이터 종속으로 인하여 완전한 병렬 수행을 쉽게 이룰 수 없다. 따라서, 본 논문은 기존의 중첩 루프 구조를 갖는 자바 프로그래밍에서 데이터 종속성 분석에 의한 종속 그래프를 구성하여 묵시적 병렬성을 검출하는 방법을 제안한다. 또한 재구성 컴파일러에 의하여 자바 원시 프로그램을 자바 프로그래밍 언어 자체에서 지원하는 다중스레드 기법으로 변환하여 병렬 시스템에서 실행하는 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 자바 프로그램에 대한 효과적인 병렬성 검출을 가능하게 하여 병렬 시스템에서 실행 시간을 단축할 수 있다.

Transform Nested Loops into MultiThread in Java Programming Language for Parallel Processing

Deuk-Young Hwang† · Young-Keun Choi††

ABSTRACT

It is necessary to find out the parallelism in the sequential Java program to execute it on the parallel machine. The loop is a fundamental source to exploit parallelism as it process a large portion of total execution time in sequential Java program on the parallel machine. However, a complete parallel execution can hardly be achieved due to data dependence. This paper proposes the method of exploiting the implicit parallelism by structuring a dependence graph through the analysis of data dependence in the existing Java programming language having a nested loop structure. The parallel code generation method through the restructuring compiler and also the translation method of Java source program into multithread statement, which is supported by the Java programming language itself, are proposed here. The performance evaluation 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 provides efficient way of exploiting parallelism by reducing manual overhead converting sequential Java program into parallel code. The execution time for the Java program as a result can be reduced on the parallel machine.

※ 이 논문은 1998학년도 삼척산업대학교 자체학술연구조성비에 의하여 연구되었음.

† 정 회 원 : 삼척산업대학교 컴퓨터과학과
†† 정 회 원 : 광운대학교 전자계산학과

논문접수 : 1998년 4월 3일, 심사완료 : 1998년 6월 1일

1. 서 론

자바(Java)는 선 마이크로시스템사(Sun Microsystems)에서 개발한 객체지향 프로그래밍 언어이다. 정보의 서비 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[5,10].

최근 GUI(Graphic User Interface)의 개발 경험을 통해 객체지향 기술이 성숙되었고, 인터넷을 통한 월드와이드웹(World Wide Web)의 실용화와 보급이 확산되었으며, 개방형 분산 클라이언트 서버 시스템의 장점이 인식되기 시작하였다. 자바는 분산 네트워크 환경의 다양한 플랫폼에서 실행 가능하기 때문에 월드와이드웹을 통한 인터넷과 컴퓨터 업계 전반에 걸쳐 커다란 변화를 일으키고 있다[16].

병렬 프로그래밍 언어의 유형은 크게 두 가지로 분류할 수 있다. 첫째는 순차 프로그래밍 언어에 자원관리 및 특정 기계의 세부사항을 명시하는 병렬 프리미티브(primitive) 추가에 의해 명시적인 병렬 언어를 만드는 방법으로 사용자는 새로운 병렬 프로그래밍 언어를 습득해야 하는 오버헤드가 있고, 병렬성의 검출을 사용자에게 요구하는 단점을 가지고 있다. 기존에 작성된 많은 자바 프로그램을 병렬 프로그램으로 변환하는 데에는 객체지향의 개념을 사용하였다 할지라도 많은 시간이 소모된다. 둘째는 순차 프로그램을 입력받아 병렬 프로그램으로 변환하는 병렬화 컴파일러에 의한 방법이 있다. 대부분의 순차 프로그램들은 루프를 처리하는데에 많은 시간을 소요하므로 속도 향상을 위해 프로그램을 변환시켜 주는 작업이 필요하다. 이 기능을 하는 재구성 컴파일러(restructuring compiler)는 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[2,3,7,9,19].

따라서 병렬 시스템에서 순차 자바 프로그램을 재사용할 수 있기 위해서는 자바 프로그램 내에 존재하는 병렬성을 찾아내는 것이 중요하다. 자바 프로그램을 병렬 시스템에서 실행할 경우 루프는 전체 수행 시간 중 많은 부분을 차지하므로 병렬성 검출의 기본이 되지만 데이터 종속으로 인하여 완전한 병렬 수행을 쉽게 이룰 수 없다. 루프의 형태는 모든 이터레이션(iteration)들 사이에 종속성이 없는 경우에 병렬 구문 DOALL로 변환하거나 서로 다른 이터레이션 사이에 종속성이 발생하는 경우에 병렬 구문 DOACROSS로 변환하여 병렬처리하는 방법이 있다[2,3,13,19].

본 논문은 기존에 작성된 중첩 루프 구조를 갖는 자바 프로그래밍에서 데이터 종속성 분석에 의한 종속 그래프를 구성하여 묵시적(implicit) 병렬성을 검출하는 방법을 제안한다. 또한 재구성 컴파일러에 의하여 자바 원시 프로그램을 자바 프로그래밍 언어 자체에서 지원하는 다중스레드(multithread) 기법으로 변환하여 병렬 시스템에서 실행하는 방법을 제안한다. 스레드(thread) 문장으로 변환된 프로그램에 대해 루프의 반복계수와 스레드 수를 매개변수로 하여 성능 분석을 하였다. 재구성 컴파일러에 의한 장점은 사용자의 병렬성 검출에 대한 오버헤드를 줄이고, 순차 자바 프로그램에 대한 효과적인 병렬성 검출을 가능하게 하여 병렬 시스템에서 실행 시간을 단축할 수 있다.

2. 데이터 종속성

2.1 데이터 종속의 종류

본 논문에서 사용한 다중 중첩 for 루프 모델은 다음과 같으며 각 루프마다 인덱스 변수의 경계값과 증분값은 정규화되었다고 가정한다.

```
for (i1 = L1; i1 < U1; N1) {
  for (i2 = L2; i2 < U2; N2) {
      ......
      for (in = Ln; in < Un; Nn) {
Si :  A(f1(i1,···,in),···,fm(i1,···,in)) = ·····
Sj        ·····    = A(g1(i1,···,in),···,gm(i1,···,in))
      }
  }
}
```

위에서 두 문장 Si와 Sj 사이에 종속성을 찾기 위해서는 배열변수명 A에 대한 각각의 첨자를 비교·분석하여야 한다. 위 루프에서 종속의 유무는 n개의 다음 방정식을 동시에 만족하는 $\alpha = (i1, i2, \cdots, in)$와 $\beta = (i1', i2', \cdots, in')$가 존재하는가에 따른다.

$$f_i(\alpha) = g_i(\beta) \quad (\forall_i,\ 1 \le i \le m)$$

즉, 두 문장 $S_i(\alpha)$와 $S_j(\beta)$ 사이의 데이터 종속성은 $S_i \delta S_j$ 형태로 표현하여 문장 S_j는 S_i에 종속된다고 한다. 데이터 종속성은 세 가지 유형으로 분류하여 사용한다. 첫 번째 흐름 종속성(δ^t)은 문장 S_i에서 변수 A가 정의(define)되고 문장 S_j에서 A가 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 A가 사용하고 문장 S_j에서 A가 정의되며 S_i가 S_j 이전에 수행된다. 또한 데이터 종속성은 세 가지 유형 중 하나만 만족하면 종속이 존재한다고 한다[2,3,7,9,13,19].

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 이터레이션(iteration)들 사이에서는 종속성이 존재하지 않고, 동일한 이터레이션 내에서만 종속성이 존재하는 경우이다. 즉 같은 이터레이션 내에서 문장 S_i에서 변수 A가 정의되고 문장 S_j에서 A가 사용되는 경우에 S_j는 S_i에 대하여 LID가 존재한다. LCD는 루프내의 서로 다른 이터레이션들 사이에 종속성이 존재하는 경우이다. 즉 루프의 한 이터레이션에서 문장 S_i에서 변수 A가 정의되고, 다른 이터레이션에서 문장 S_j에서 A가 사용되는 경우에 S_j는 S_i에 대하여 LCD가 존재한다.

따라서 루프의 형태는 모든 이터레이션들 사이에 종속성이 없는 경우와 다른 이터레이션 사이에 종속성이 발생하는 경우가 있다. 루프를 펼쳤을 때(unrolled) 모든 문장 각각을 인스턴스(instance)라 하며 특히, 종속의 근원이 되는 인스턴스를 source 인스턴스, 종속이 되는 인스턴스를 sink 인스턴스라 한다.

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

2.2 방향 벡터와 거리 벡터

Wolfe에 의해 제안한 데이터 종속과 관련된 개념으로 방향 벡터(direction vector)가 있다[2]. 레벨 n의 중첩 루프에서 방향 벡터(Ψ)는 다음과 같이 정의한다.

$\Psi = \langle \Psi_1, \Psi_2, \ldots, \Psi_n \rangle$ (단, $\Psi_i \in \{<, =, >, \le, \ge, \ne, *\}$)

방향 벡터는 i_j Ψ_j i_j'인 조건에서 부등식 검사를 적용하여 모든 $1 \le j \le n$에 대해 α와 β를 만족하는 값이 존재하는가를 검사한다.

거리 벡터(distance vector)는 같은 기억장소를 참조하는 두 첨자를 비교하므로서 루프 이터레이션에 대한 실제 거리를 알려준다. 즉, n 단계의 중첩 루프에서 j번째 루프 첨자에 대하여 이터레이션 α내의 어떤 문장으로부터 이터레이션 β내의 문장으로의 흐름 종속성이 존재할 때 같은 첨자에 대한 이터레이션 값의 차이를 거리 벡터(Φ_i)라 하고 다음과 같이 정의할 수 있다.

$\langle \Phi_1, \Phi_2, \ldots, \Phi_n \rangle$ ($\Phi_i = \beta_i - \alpha_i$, $(1 \le i \le n)$)

여기서, 실제 종속 거리(ϕ_{ij})는

$\phi_{ij} = \sum_{r=1}^{n} (\Phi_r \sum_{m=r+1}^{n} N_m)$이고, 두 문장 S_i와 S_j 사이의 종속에 대한 전체 이터레이션 수를 나타낸다.

거리 벡터의 값이 양수이면 방향 벡터는 '<'로 표현하고, 방향 벡터의 값이 음수이면 방향 벡터는 '>'로 표현하고, 거리 벡터의 값이 영이면 방향 벡터는 '='로 표현한다.

2.3 데이터 종속성 그래프

프로그램에서 데이터 종속성을 표현하는 그래프의 중간표현 형태로 데이터 종속성 그래프(data dependence graph)가 있다. 데이터 종속성 그래프는 G = (V, E) 형태의 방향성 그래프로 표현한다. 여기서 V는 그래프 상의 노드로서 한 문장이나 기본 블록(basic block) 단위이고, E는 두 노드 사이의 데이터 종속을 표현하는 방향성 에지이다[1,2,3].

따라서 다음과 같이 데이터 종속성 그래프를 정의한다.

1. 데이터 종속성 그래프는 G = (V, E) 형태의 방향성 그래프로 표현한다.
2. V = {S_1, S_2, ..., S_n}는 그래프 상의 노드로서 한 문장 단위이다.
3. E = {e_{ij} = (S_i, S_j) | S_i,S_j∈V}는 두 노드 사이의 데이터 종속을 표현하는 방향성 에지이다.

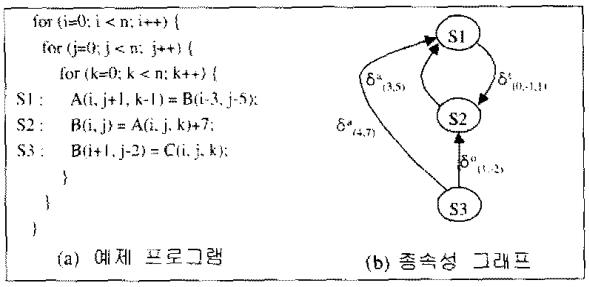
본 논문에서는 다음과 같은 단계로 데이터 종속성 그래프를 구성한다.

① 2.1절에서 기술한 방법을 이용하여 중첩 루프에서 노드들 사이의 데이터 종속성 관계를 추출한다.

② 2.2절에서 기술한 방법을 이용하여 데이터 종속성에 대한 거리 벡터를 추출한다.

③ 추출한 데이터 종속성 정보를 갖고 정의에 따라 노드들 사이를 방향성 에지로 연결한다.

예를 들면 (그림 1)과 같이 데이터 종속성 그래프를 구성한다.



(그림 1) 데이터 종속성 그래프
(Fig. 1) Data dependence graph

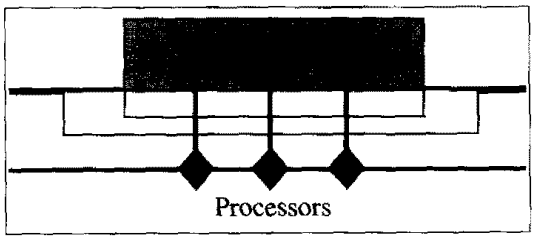
(그림 1)의 예제 프로그램에서 배열 A의 첨자에 대한 거리 벡터는 Φ_1 = i - i = 0, Φ_2 = j-(j+1) = -1, Φ_3 = k-(k-1) = 1이다. 따라서, 배열 A의 종속에 대한 종속 거리는 Φ = (0,-1,1)이고, 방향 벡터는 Ψ = (=, <, >)이다.

3. 중첩 루프 구조의 병렬성 검출

3.1 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(langu-age level) 지원을 제공하지 못한다. 하지만 자바에서는 언어 차원에서 하나의 주소 공간(address space)을 공유하는 여러 개의 스레드들이 병행적으로 실행할 수 있는 다중스레드를 지원한다[4,6,8,11].

본 논문에서는 병렬 프로그램의 각 프로세서와 제어 흐름을 나타내기 위해 다중스레드를 사용한다. (그림 2)는 다중 프로세서 시스템에서 서로 다른 프로세서가 다른 스레드를 실행하는 상태를 나타낸다. 하나 이상의 프로세서를 갖는 컴퓨터는 동시에 여러 개의 프로세서를 실행하며, 다중스레드는 사용자가 하드웨어의 병렬성을 추구하는데 효과적인 방법이다. 여러 개의 스레드들은 사용자의 특별한 입력이 없이 동시에 다른 프로세서 상에서 실행할 수 있다.

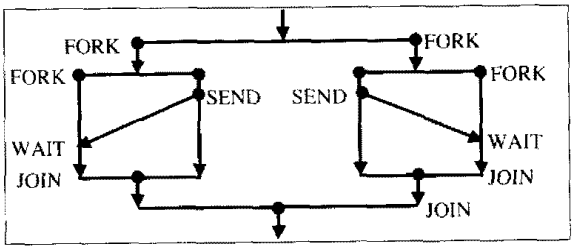


(그림 2) 여러 개의 프로세서에서 실행하는 스레드
(Fig. 2) Different threads running on different proc-essors

자바에서 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() 메소드를 호출한 다음의 코드를 비동기적으로 계속 실행한다. 또한 자바에서 스레드 사이의 통신을 위해 wait(), notify(), notifyAll() 메소드를 제공한다. wait() 메소드는 어떤 조건이 발생할 때까지 스레드를 대기 상태로 만드는 메소드이고, notify() 메소드는 현재 대기 상태에 있는 스레드 중 하나를 실행(run) 상태로 만드는 메소드이고, notifyAll() 메소드는 현재 대기 상태에 있는 모든 스레드를 실행 상태로 만드는 메소드이다. wait(), notify(), notifyAll() 메소드는 Thread 클래스가 아니라 Object 클래스에 정의되어 있기 때문

에 모든 클래스에 상속될 수 있다. 또한 스레드는 join() 메소드를 호출하여 다른 스레드가 종료될 때까지 대기할 수 있다[12,14,15,16,17].

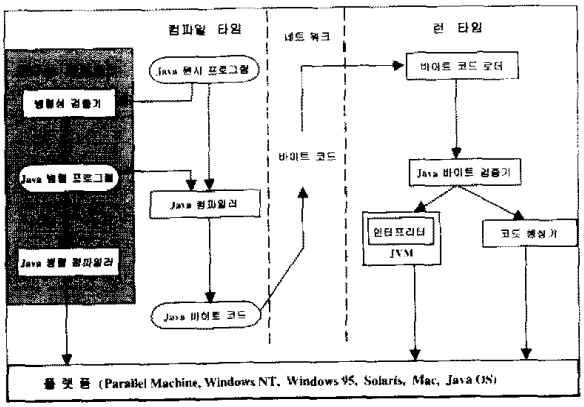
본 논문에서는 자바 순차 프로그램을 다중 프로세서를 가진 시스템에서 실행 가능하도록 스레드 프로그램으로 변환하며, (그림 3)과 같이 실행 순서를 FORK/JOIN 구조로 구현한다.



(그림 3) FORK/JOIN 구조
(Fig. 3) FORK/JOIN construct

3.2 재구성 컴파일러의 구조

본 논문은 재구성 컴파일러에 의하여 자바 원시 프로그램을 자바의 다중스레드 기법으로 변환하여 병렬 시스템에서 수행하는 알고리즘을 제안한다. (그림 4)의 재구성 컴파일러에서 병렬성 검출기는 자바 순차 프로그램에서 묵시적인 병렬성을 포함하고 있는 여러 개의 중첩 루프 구조에 대한 데이터 종속성 분석을 수행한다. 데이터 종속성 분석에 의하여 만약 루프의 모든 이터레이션이 독립적이다면, 즉 데이터 종속이 루프에 연관되지 않는 LID 종속이 존재하거나 이터레이션내의 두 문장 사이에 전혀 서로 다른 변수를 사용할 때 이



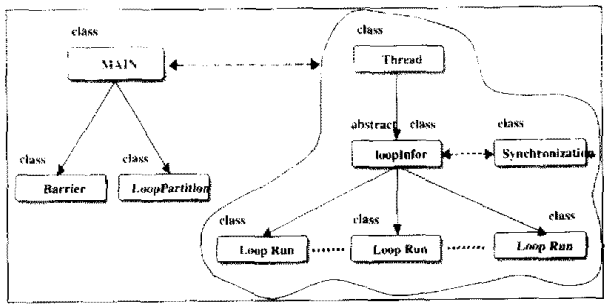
(그림 4) 자바의 컴파일과 수행과정
(Fig. 4) Compile-time and run-time of Java

[illegible] 다중스레드 의미를 갖는 병렬 구문 DOALL 구조의 형태로 변환한다. 비록 데이터 종속이 루프에 연관되는 LCD일지라도 동기화 문장을 삽입하여 다중스레드 의미를 갖는 병렬 구문 DOACROSS 루프의 형태로 변환한 후 이터레이션을 부분적으로 실행하여 묵시적인 병렬성을 얻을 수 있다[2,3,7,13].

따라서 변환된 자바의 병행 프로그램은 다중 프로세서를 가진 병렬 자바 컴파일러에 의하여 자바 언어의 중간코드인 바이트 코드로 번역하여 다양한 컴퓨터 플랫폼에서 수행함으로써 묵시적인 병렬성을 얻을 수 있다.

3.3 중첩 루프 구조의 다중스레드로 변환

본 논문은 자바 언어에서 중첩 루프 구조를 자바의 다중스레드 구문으로 변환하기 위하여 (그림 5)와 같이 클래스 계층구조를 구성한다. 자바에서 스레드는 자바 API의 java.lang 패키지의 Thread 클래스를 사용하여 생성한다[11,15]. (그림 5)에서 MAIN 클래스는 LoopPartition 클래스를 호출하고, 필요한 경우에는 배리어(barrier) 동기화를 수행하고, 병렬 수행 부분인 Thread 클래스를 호출한다. 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. Synchronization 클래스는 여러 개의 프로세서가 공유 데이터를 접근할 때 동기화를 부여한다. LoopPartition 클래스는 다중 프로세서 수에 따라 중첩 루프의 모든 이터레이션을 블록 스케줄링 방법을 이용하여 프로세서에 할당한다. Barrier 클래스는 모든 프로세서가 실행을 끝낼 때까지 대기한다.



(그림 5) 클래스 계층 구조
(Fig. 5) Class Hierarchy

3.3.1 프로세서 할당

공유메모리 구조를 갖는 병렬 시스템에서 병렬 루프는 루프의 서로 다른 이터레이션을 다중 프로세서에 할당하여 병렬로 실행한다. 자바에서는 루프의 서로 다른 이터레이션을 다중스레드에 할당하여 FORK/JOIN 형태의 병렬성을 이용하여 실행할 수 있다. 본 논문에서는 모든 이터레이션이 똑같은 작업을 수행하는 정적 할당 블록 스케줄링 방법을 이용하여 병렬로 수행한다 [9].

(알고리즘 1) 프로세서 할당 클래스
(Algorithm 1) Processor Allocation Class

```
Input : lower-upper bound
Output : Initial of lower-upper bound

procedure BoundInitial(low, upper, lowhigh)
 /* 중첩 루프의 하한값과 상한값을 초기화 */
for every loopNode ∈ {LOOP} begin
    lowhigh[i][j] = loopNode.lower
    lowhigh[i+1][j] = loopNode.upper
    j += 2
end
end /* end of BoundInitial class */

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

procedure BoundPartition(lowhigh, ProcessorN,
                         forCount, lbub)
 /* 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
 /* 프로세서 수에 따라 중첩 루프의 이터레이션을 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 BoundPartition class */
```

(알고리즘 1)은 각 루프의 경계값을 초기화는 BoundInitial 클래스와 루프를 분할하는 BoundPartition 클래스로 구성한다. BoundInitial 클래스는 각 중첩 루프의 레벨 1에 대한 하한값과 상한값을 입력으로 받아 루프 분할을 하여 2차원 배열 lowhigh 변수에 저장한다. BoundPartition 클래스의 수행 과정은 lowhigh 변수에 저장된 초기값, 프로세서 수, 반복 구문 for 문의 수를 입력으로 받아서 프로세서 수에 따른 2차원 배열에 루프 이터레이션을 분할하는 프로세서 할당 알고리즘이다. 먼저 다음의 식과 같이 프로세서 수에 따라 정적 할당하는 블록 크기(blocksize)를 결정한다.

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

다음으로 모든 중첩 루프 문장을 프로세서 수에 따라 하한값과 상한값을 일정하게 분할하여 2차원 배열 lbub 변수에 저장한다.

본 논문에서는 설명을 위하여 (그림 6)과 같이 중첩 루프를 갖는 NestedLoop 클래스의 예제 프로그램이 존재한다고 가정한다. (알고리즘 1)에 의해 (그림 6)의 자바 예제 프로그램을 2개의 프로세서를 가진 병렬 시스템에 정적 할당한다고 가정할 경우에 첫 번째 프로세서에 할당되는 루프의 하한값과 상한값은 0과 50이고, 두 번째 프로세서에 할당되는 루프의 하한값과 상한값은 50과 100이 할당된다.

```
class NestedLoop {
     public static void main(String args[]) {
        int N = 100;
        double[][] a,b = new double[N][N];
/*L1*/    for (int i=0; i < N; i++)
/*L2*/       for (int j=0; j < N; j++) {
/*S1*/          a[i][j] = a[i][j] * 2;
/*S2*/          b[i][j] = a[i+2][j] + b[i][j];
              }
     }
}
```

(그림 6) 예제 프로그램
(Fig. 6) Example Program

3.3.2 동기화 방법

일반적으로 순차 프로그램에서 각 이터레이션은 선행 이터레이션에 의하여 생성된 데이터를 사용하여 올바른 해를 생성한다. 그러나 공유메모리 구조를 갖는 병렬 시스템에서 병렬 프로그램의 루프는 서로 다른 이터레이션을 다중 프로세서에 할당하여 병렬로 실행하므로 올바른 해를 생성하지 않을 수도 있다. 또한, 자바

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, 자바에서는 동기화를 위해 모니터의 잠금(lock)을 사용한다. 동기화는 두 개의 스레드를 상호배제로 수행하게 된다[14,15].

3.3.2.1 동기화 클래스

다중스레드 환경에서 사용 가능한 클래스를 만들기 위해서 메소드들은 synchronized로 선언되어야 한다. 스레드가 하나의 객체 상에서 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 블록 된다.

본 논문은 (알고리즘 2)와 같이 병렬 구문 DOACROSS 형태의 루프에서 동기화를 요구하는 변수를 입력으로 받아서 동기화 변수의 특정 비트를 잠금하는 send 메소드와 동기화 변수의 특정 비트가 잠금될 때까지 블록킹하는 wait 메소드를 사용한다.

(알고리즘 2) 동기화 클래스
(Algorithm 2) Synchronization Class

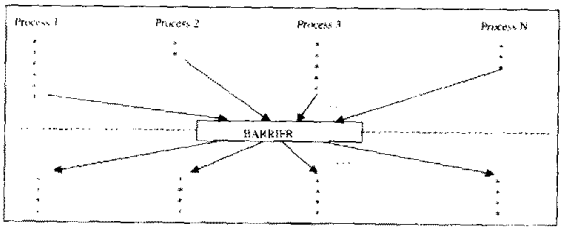
```
Input : Synchronization Variable
Output : Lock or Unlock

procedure class Sync(low, high)
lock = new boolean[high + low]
 /* 1. Send Synchronous Method */
synchronized send(i) begin
          lock[i] = TRUE
          notifyAll()
end
 /* 2. Wait Synchronous Method */
synchronized wait(j) begin
          if ((j-low) >= 0) begin
              while (!lock[j])
                    wait()
          end
end
end /* end Synchronization procedure class */
```

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

3.3.2.2 배리어 클래스

(그림 7)과 같이 병렬 시스템에서 각 프로세서는 어느 시점에서 모든 프로세서가 도달할 때까지 일시적으로 대기하는 배리어 동기화 문장이 필요한 경우가 있다. 배리어 문장을 실행하는 첫 번째 프로세서는 모든 다른 프로세서가 도달할 때까지 대기하다가 모든 프로세서가 배리어 문장에 도달하면 해제함으로써 다음 작업을 병렬로 계속 실행하여 결과를 얻을 수 있다.



(그림 7) 배리어 동기화
(Fig. 7) Barrier Synchronization

본 논문은 (알고리즘 3)과 같이 자바 언어에서 실행중인 스레드와 프로세서 수를 입력으로 받아서 실행하는 모든 스레드가 종료할 때까지 대기하는 join 메소드를 이용하여 배리어 동기화 문장을 구현하였다.

(알고리즘 3) 배리어 클래스
(Algorithm 3) Barrier Class

```
Input : Running Thread, Processor Number
Output : Thread Join

procedure class Barrier(loopInfor[] loopThread,
                        ProcessorN)
begin
   for every i = [1..ProcessorN] begin
      if (loopThread[i] != NULL)
          loopThread[i].join()
   end
end /* end of Barrier class */
```

3.3.3 주 프로그램 변환

주 프로그램 변환은 main 클래스 생성, 중첩 루프 구조의 추상화 클래스 생성, 원시 프로그램의 루프 구조 변환 생성의 세부분으로 구성한다.

3.3.3.1 main 클래스

main 클래스 생성은 (알고리즘 4)와 같으며, 다음과 같은 과정을 수행한다. 먼저 프로세서 수에 대한 변수 선언과 calcLohi 함수 호출을 통해 for 문장 수를 생성한다. 중첩 루프의 하한값과 상한값을 초기화하기 위한 2차원 배열 lowhigh 변수의 메모리를 할당하고, 프로세서 할당에 필요한 이터레이션 하한값과 상한값을 저장하기 위한 2차원 배열 lbub 변수를 생성한다. 심볼 테이블로부터 원시 프로그램의 선언문 부분을 생성한다. 종속성 검사에 의한 동기화가 필요한 문장의 수를 생성한다. 각 중첩 루프의 외부 경계값을 초기화하는 BoundInitial 클래스와 중첩 루프의 정적 스케줄링 호출을 위한 BoundPartition 클래스를 생성한다. 동기화 객체 Sync를 생성하여 동기화 변수인 비트 배열의 메모리를 할당하기 위한 문장을 생성한다.

마지막 단계로 루프 구조 생성 및 스레드 생성 호출 부분은 프로그램 내에 중첩 for 문장의 개수가 하나일 때와 n개일 때로 나누어 처리한다. 중첩 for 문장의 개수가 하나일 때는 loopSubscriptedDependenceC-heck 프로시저를 호출하여 종속성이 존재하면 중첩 루프의 내부와 외부를 교환하거나 동기화 문장을 삽입한 후, genLoop 프로시저를 호출하여 병렬 실행문을 생성하고 프로세서 개수만큼의 스레드 객체를 생성하여 병렬로 실행한다. 중첩 for 문장의 개수가 하나 이상일 때는 인접한 중첩 for 루프 L_i와 L_j 사이에 종속이 존재하지 않는다면 두 개의 중첩 루프는 하나의 클래스 내에서 실행할 수 있다는 것을 의미하기 때문에 병합하여 프로세서 개수만큼의 다중스레드를 생성하여 병렬로 실행한다. 예를 들면 loopBlock[1]과 loopBlock[2]가 존속이 존재하지 않는다면 loopBlock은 {1, 2} 정보를 갖는다.

또한 인접한 중첩 for 문장에 대한 하한값과 상한값을 병합한다.

즉, 첫 번째 for 문장 lbub[1]의 하한값과 상한값이 {lbub[0][0], lbub[0][1]}이고, 두 번째 for 문장 lubu[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 중첩 for 루프 L_i와 L_j 사이에 종속이 존재한다면 중첩 루프 L_i와 L_j는 서로 다른 클래스에서 실행하기 위한 루프 구조 생성과 스레드 호출 부분을 삽입하고 루프 L_i와 L_j 사이에 배리어 동기화 문장을 삽입하여 프로세서 개수만큼의 다중스레드를 생성하여 병렬로 실행한다. 예를 들면 loopBlock[1..3] = {1, 2, 3}에 대해 loopBlock[1]과 loopBlock[2]는 종속이 존재하지 않고 loopBlock[3]은 loopBlock[2]에 대해 종속이 존재한다면 loopBlock[1..2] = {1, 2}는 동일한 클래스에서 실행할 수 있지만 loopBlock[3] = {3}은 동일한 클래스에서 실행할 수 없으므로 루프를 분할하여 배리어 동기화 문장을 삽입해야 한다.

이와 같은 방법으로 (알고리즘 4)에 의해 (그림 6)의 자바 예제 프로그램의 main 클래스를 생성하면 다음과 같다.

```
class Main {
  static NestLoop1[] t;
  public static void main(String args[]) {
   /* 프로세서 수, for 문장의 수, 동기화 문장 수, 루프의 하한값과 상한값 초기화 */
   /* 사용한 모든 배열의 메모리 할당 */
   /* 알고리즘 1 호출 */
     for (int i = 0; i < p_no; i++) {
        t[i] = new NestLoop1(lb_ub[i][0],
                lb_ub[i][1], N, a, b, P);
        t[i].start();
     }
  }
}
```

(알고리즘 4) 주 프로그램 클래스
(Algorithm 4) Main Program Class

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

procedure class MainGenerate()
  generate(ProcessorN, forCount = calcLoHi()) /* 프로세서 수와 중첩 루프 수를 생성 */
  generate(lowhigh = new [2][forCount * 2])
  generate(lbub = new [ProcessorN][forCount * 2])
  generate(declarlation(SYMBOL)) /* 심볼 테이블로부터 원시 프로그램의 선언문 생성 */
  /* 종속성 검사에 의한 동기화 문장 수를 구함 */
  generate(sync no = loopSubscriptedDependenceCheck());
  generate(call BoundInitial(lowhigh, forCount)) /* 스케줄링 부분 호출 문장 생성 */
  generate(call BoundPartition(lowhigh, ProcessorN, forCount, lbub))
  generate(Sync[] P = new Sync(sync_no)) /* 동기화 문장 메모리 할당 */
  generate(for i=[0..sync_no] step 1)
    generate(P[i] = new sync(lowMin, highMax))
```

```
/* 루프 블록 생성 및 스레드 생성 호출 */
if (forCount == 1) begin /* 단일 중첩 for 문장인 경우 */
   call loopSubscriptedDependenceCheck(currentFor[1])
   call genLoop(lbub[0], loopBlock[1], P, 1)
   generate(for j = {0..ProcessorN})
      generate(call loop+(0)(lbub[0], UseDefSet(SYMBOL), P)
      generate(loop+(0)[j], start)
end /* end of if (forCount = 1) */
for (i = 1 ; i < forCount; i++) begin /* 2개 이상의 중첩 for 문장인 경우 */
   /* 중첩 루프에 대해 종속성 검사를 수행 */
   call loopSubscriptedDependenceCheck(currentFor[i])
   call loopSubscriptedDependenceCheck(currentFor[i+1])
   /* 인접한 중첩 루프 사이에 종속이 없을 경우에 루프 병합 */
   if (call adjacentNestedloopDependencecheck() = TRUE) begin
       loopBlock = loopBlock ∪ {{loopBlock[i]} ∪ {loopBlock[i+1]}}
       lhMrg = lhMrg ∪ {{lbub[i]} ∪ {lbub[i+1]}}
   end
   else begin /* 인접한 중첩 루프 사이에 종속이 존재할 경우 */
       call genLoop(lhMrg, loopBlock, P, i)
       generate(for j = {0..ProcessorN})
             generate(call loop+(i)(lhMrg, UseDefSet(SYMBOL), P)
             generate(loop+(i)[j], start)
       generate(call Barrier())
       generate(call genLoop(lbub[i+1], loopBlock, P, i+1))
       generate(for j = {0..ProcessorN})
             generate(call loop+(i+1)(lbub[i+1], UseDefSet(SYMBOL), P)
             generate(loop+(i+1)[j], start)
       i++
       loopBlock = Φ
       lhMrg = Φ
   end
end
if (loopBlock != Φ) begin /* 인접한 중첩 루프 사이에 종속이 없는 경우 */
       call genLoop(lhMrg, loopBlock, P, i)
       generate(for j = {0..ProcessorN})
             generate(call loop+(i)(lhMrg, UseDefSet(SYMBOL), P)
             generate(loop+(i)[j], start)
       end
end /* end MainGenerate procedure class */
```

(알고리즘 5)에서 calcLoHi 프로시저는 중첩 루프에 속하는 모든 loopNode에 대하여 중첩 for 문장의 개수와 동기화 변수인 비트 배열에 메모리를 할당하기 위한 루프 경계값을 계산한다. loopSubscriptedDependenceCheck는 종속성을 검사하는 프로시저이며, 먼저 모든 중첩 루프의 레벨에 따라 종속 거리를 계산하여 종속성 개수와 종속성 위치를 구한다. 만약 데이터 종속성이 외부 첨자에 대하여 존재하고 내부 첨자에는 종속성이 존재하지 않는다면 루프 교환을 수행한다. 병렬 시스템에서 루프 교환을 하는 이유는 적은 수의 FORK와 JOIN을 생성하여 병렬성을 향상시키고자 하는데 목적이 있다. 만약 데이터 종속성이 외부 첨자와 내부 첨자 전부 존재한다면 외부 첨자에 대하여 동기화 문장 send, wait와 종속 거리를 삽입하여 병렬 구문 DOACROSS 형태의 루프로 병렬 실행할 수 있다.

(알고리즘 5) 종속성 검사
(Algorithm 5) Dependence Test

```
procedure calcLoHi() /* 중첩 for 문장 개수와 동기화 변수인 비트 배열을 할당하기 위한 크기를 얻는다. */
  for every loopNode ∈ {LOOP} begin
    forCount++
lowMin = min(lowMin, loopNode.lower)
    highMax = max(highMax, loopNode.upper)
  end
  return forCount
end /* end procedure calcLoHi */

procedure loopSubscriptedDependenceCheck()
begin
    /* 중첩 루프의 종속성 개수를 구한다. */
  for every i = {1..Depth} begin
    for every α ∈ for(i).Distance  begin
      if (α ≠ 0) then DependenceCheck[i]++
    end
  end
    /* 중첩 루프의 종속성 위치를 구한다. */
  for every i = {1..Depth} begin
    if (DependenceCheck[i] ≠ 0) begin
      subScriptedDependence++
      pos = i
    end
  end
    /* 중첩 루프의 교환 */
  if (subScriptedDependence = pos = 1) begin
    for every i={2..Depth} begin
      if (for[i].lower = for[1].lower) &
            (for[i].upper = for[1].upper) begin
        loopInterchange(for[i], for[1])
        break
      end
    end
  end
    /* 동기화 문장 삽입 */
  if (DependenceCheck[1] ≠ 0) begin
    for every (α ∈ for(1).Distance) ≠ 0 ) begin
      insertSourceCode(send(subScripted(1))
      insertSourceCode(wait(subScripted(1) + α)
      sendWaitCount++
    end
  end
  return sendWaitCount
end
```

3.3.3.2 추상화 클래스

중첩 루프 구조의 추상화 클래스 생성은 (알고리즘 6)과 같으며, 다음과 같은 과정을 수행한다. 먼저 java.lang.Thread의 상위 클래스에 정의되어 있는 메소드를 계승받아 abstract 키워드를 붙여 추상화 클래스 loopInfor를 생성한다. 중첩 루프 구조에서 루프 병합한 하한값과 상한값, 동기화 변수의 선언을 생성한다. 매개 변수로 루프의 하한값과 상한값, 동기화 변수를 갖는 loopInfor 메소드를 생성한다. 마지막으로 클래스

: 매개변수로 입력받은 배열의 하한값과 상한값을 멤버 데이타에 할당하는 문장을 생성한다.

(알고리즘 6) 중첩 루프 구조의 추상화 클래스
(Algorithm 6) Abstract Class of Nested Loop Structure

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

procedure class genloopInfor(argument_list ...)
  generate(abstract + class + loopInfor + exter
          + 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 */
```

(알고리즘 6)에 의해 (그림 6)의 자바 예제 프로그램에 대한 중첩 루프 구조의 추상화 클래스를 생성하면 다음과 같다.

```
abstract class loopInfor extends Thread {
  /* run() 메소드에서 사용할 중첩 루프의
  하한값과 상한값, 동기화 변수를 선언 */
 public loopInfor(int low, int high, Sync P)
  { /* 매개 변수로 받은 값을 멤버
       데이터에 할당 */ }
}
```

3.3.3.3 원시 프로그램의 루프 변환

원시 프로그램의 루프 변환은 (알고리즘 7)과 같으며, 다음과 같은 과정을 수행한다. 추상화 클래스 loopInfor를 상위 클래스로 하는 loop+i의 클래스를 생성한다. 중첩 루프 병합에 의해 생성된 루프내에서 사용되거나 정의된 변수의 선언을 생성한다. 메소드 loop+i의 매개변수로 2차원 배열의 하한값과 상한값 리스트, 루프에서 정의되거나 사용된 변수, 1차원 배열의 동기화 변수를 인수로 받아 생성자 스레드 객체를 생성한다. 이 생성자는 super() 메소드를 이용하여 상위 클래스의 생성자를 호출하여 매개변수로 받은 값을 멤버 데이터에 할당하고 스레드 이름을 인수로 전달한다. 객체가 생성된 후 객체상의 start() 메소드가 호출될 때 스레드를 실행하는 run() 메소드를 생성한다. 마지막으로 중첩 루프에 속하는 모든 중첩 for 루프에 대하여 외부 루프는 정적 할당으로 생성된 하한값과 상한값을 배정하고 내부 루프와 실행 문장은 원시 프로그램의 내용을 복사하여 생성한다.

(알고리즘 7)에 의해 (그림 6)의 자바 예제 프로그램에 대한 루프 변환 클래스를 생성하면 다음과 같다. (그림 6)의 NestedLoop 클래스는 외부 첨자 i에 대하여 종속이 존재하고 내부 첨자 j에 대해서는 종속이 존

(알고리즘 7) 루프 변환 클래스
(Algorithm 7) 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))  /* lhMrg내에 사용되거나 정의된 변수 선언 */
    generate(public + className(loop+i) + declaration(lhMrg, declaration(LoopMrg), Sync[] P)
    generate(this.declaration(LoopMrg) ← declaration(LoopMrg))
    generate(public + void + run())
    for every loopNode ∈ {LOOP} begin
          generate(for i = lhMrg[i][0]; i < lhMrg[i][1]; i++) begin
                    depth
                     Σ generate (for subScripted(k)=for(k).lower; for(k).upper)
                    k=1
                    source=getSourceCode(loopNode)
          end
    end
end /* end genLoop class */
```

재하지 않으므로 중첩 루프 교환을 수행한다.

```
class NestLoop1 extends loopInfor {
        /* 루프 내에서 사용되거나 정의된
           변수의 선언 */
    public NestLoop1(int low, int high,
 int N, double[] a, double[] b, Sync[] P) {
      super(low, high, P);
      /* 매개 변수로 받은 값을 멤버
         데이터에 할당 */
    }
    public void run() {
       for(int j = low; j < high; j++)
           for(int i = 0; i < N; i++) {
       /* 동기화 문장이 삽입된 원시
       프로그램의 실행 문장 부분을 복사 */
           }
    }
}
```

4. 적용 및 분석 결과

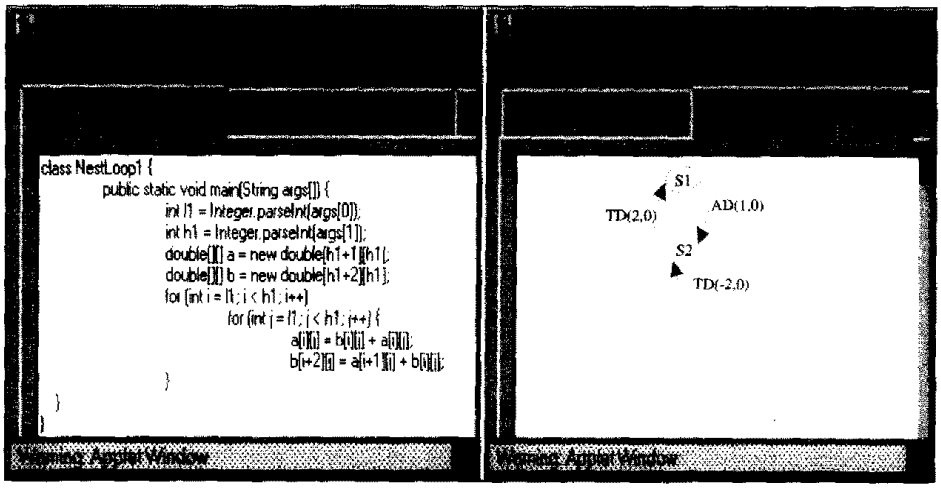
본 논문의 프로그래밍 환경은 선 솔라리스(Solaris) 2.5.1 운영체제를 가지고 있는 선 Ultra SPARC상에서 JDK1.1.5 Final 버전을 적용하여 분석하였다. 또한, 선 마이크로시스템사에서 개발한 자바 워크샵(WorkShop) 1.0 버전을 이용하여 중첩 루프 구조를 갖는 자바 원시 프로그램을 다중스레드 프로그램으로 구현하였다. 자바 워크샵은 자바 개발 환경에 통합된 것으로 썬' 와이드캠의 룸메이저를 작성하거나 자바 기술에 기반을 둔 응용 프로그램 개발에 사용하고 있다. 자바 워크샵은 GUI 환경 상에서 Portfolio Manager, Project Manager, Source Editor, Build Manager, Visual Java, Source Brower, Project Tester, Online Help 등의 도구를 지원하고 있다. 이를 이용해 사용자는 자바 원시 프로그램의 작성 및 컴파일, 디버깅 작업을 수행할 수도 있고, 프로젝트 진행을 수행할 수도 있다[18].

본 논문에서 중첩 루프의 형태는 이터레이션 사이에 종속성이 없을 때 사용하는 병렬 DOALL 루프 구조와 다른 이터레이션 사이에 종속성이 발생하는 경우에 사용하는 병렬 DOACROSS 루프 구조로 나누어 적용하였다.

4.1 DOALL 루프 구조

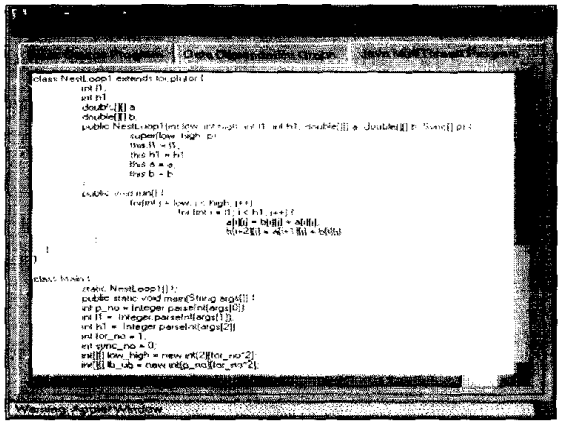
(그림 8)과 같이 하나의 중첩 for 문장 구조를 가지고 있고 종속성 분석에 의하여 외부 첨자 i에 대하여 종속이 존재하고 내부 첨자 j에 대해서는 종속이 존재하지 않는 NestLoop1 클래스의 순차 자바 프로그램이 있다고 가정하자.

(그림 8)의 종속성 그래프에서 S1과 S2는 문장 번호를 나타내고, TD와 AD는 각각 흐름 종속성과 역 종속성을 나타내며, 괄호 안의 숫자는 종속거리를 의미한다. (그림 8)의 자바 프로그램은 외부 첨자에만 종속성이 존재하므로 제안한 알고리즘에 의하여 내부 첨자와 루프 교환을 수행한다. 따라서 중첩 루프에 존재하는



(그림 8) 예제 프로그램-1
(Fig. 8) Example program-1

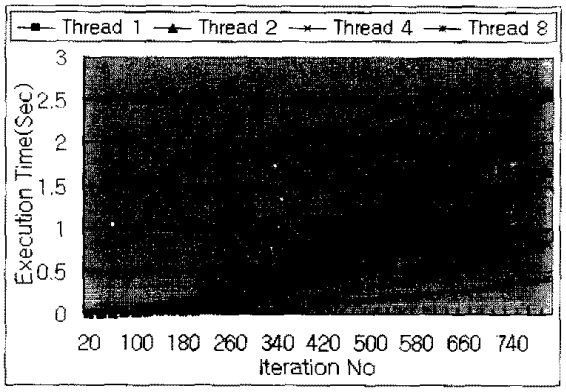
묵시적인 병렬성을 검출하여 자바 다중스레드 프로그램으로 변환하면 (그림 9)와 같다.



(그림 9) 그림 7의 스레드 프로그램
(Fig. 9) Thread program of figure 7

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

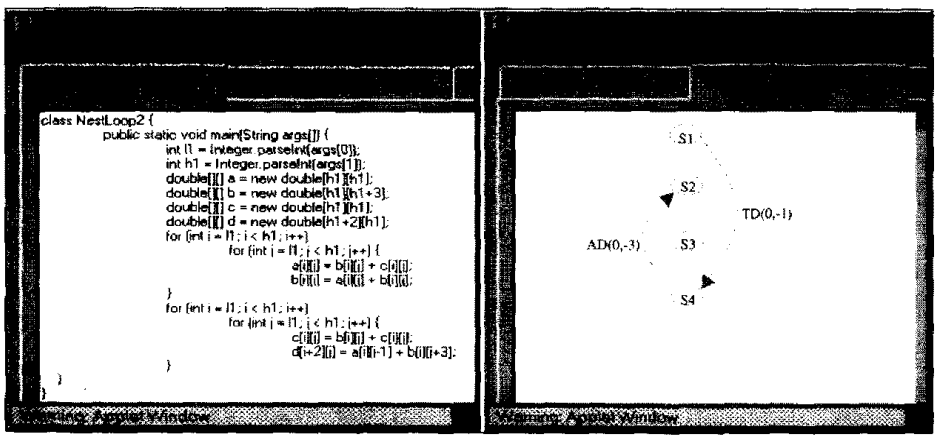
(그림 10)은 (그림 8)과 (그림 9)의 자바 프로그램에서 이터레이션 개수 N의 값 변화에 따른 실행 시간을 나타낸다. NestLoop1 클래스는 동기화 문장이 존재하지 않기 때문에 이터레이션의 수가 증가함에 따라 실행 시간이 선형적으로 증가함을 알 수 있다. 스레드가 1개인 원시 프로그램 보다 변환된 프로그램에서 스레드가 8개일 경우에 이터레이션 수가 660에서 84.5%의 성능이 향상되었다. 따라서 본 논문에서 제안한 방법에 따라 자바 원시 프로그램을 다중스레드 프로그램으로 변환하여 실행할 경우 프로세서 수가 증가함에 따라 실행 시간이 선형적으로 단축됨을 알 수 있다.



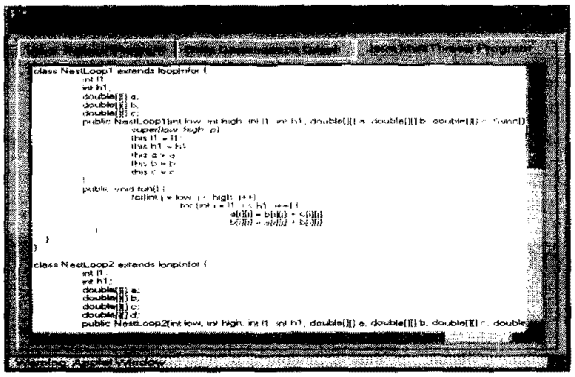
(그림 10) NestLoop1 클래스의 성능 분석
(Fig. 10) Performance analysis of NestLoop1 class

(그림 11)과 같이 2개의 중첩 for 문장 구조를 가지고 있고 종속성 분석에 의하여 각각의 중첩 for 문장 이터레이션 사이에는 종속이 존재하지 않고 서로 다른 중첩 for 문장 사이에는 종속성이 존재하는 NestLoop2 클래스가 있다고 가정하자.

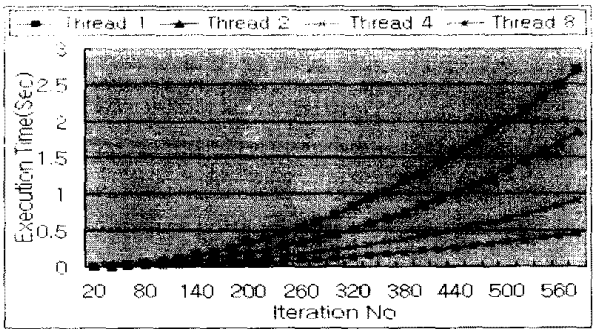
(그림 11)을 제안한 알고리즘에 의하여 묵시적인 병렬성을 검출하여 자바 스레드 프로그램으로 변환하면 (그림 12)와 같으며, 2개의 서로 다른 중첩 for 루프 사이에 종속성이 존재하므로 Main 클래스에서 배리어 동기화 문장을 삽입하고 2개의 NestLoop1과 NestLoop2 클래스에서 실행할 수 있다.



(그림 11) 예제 프로그램-2
(Fig. 11) Example program-2



(그림 12) 그림 10의 스레드 프로그램
(Fig. 12) Thread program of figure 10



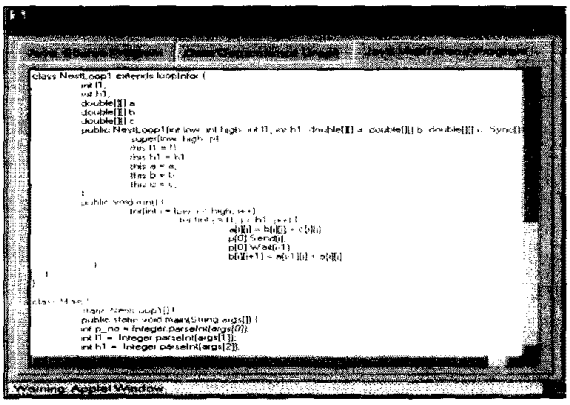
(그림 13) NestLoop2 클래스의 성능 분석
(Fig. 13) Performance analysis of NestLoop2 class

(그림 13)은 (그림 11)과 (그림 12)의 자바 프로그램에서 이터레이션 개수 N의 값 변화에 따른 실행 시간을 나타낸다. (그림 12)는 (그림 9)와 같이 이터레이션 사이에 동기화 문장이 존재하지 않기 때문에 이터레이션의 수가 증가함에 따라 실행 시간이 선형적으로 증가함을 알 수 있다. 따라서 본 논문에서 제안한 방법에 따라 자바 원시 프로그램을 다중스레드 프로그램으로 변환하여 실행할 경우 프로세서 수가 증가함에 따라 실행 시간이 선형적으로 단축되며, 두 개의 루프 사이에 배리어 동기화 문장으로 인한 약간의 오버헤드가 있음을 알 수 있다.

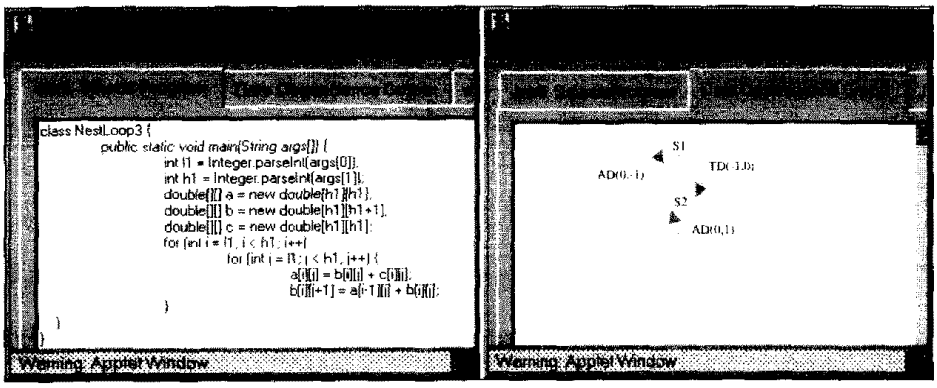
4.2 DOACROSS 루프 구조

(그림 14)와 같이 하나의 중첩 for 문장 구조를 가지고 있고 종속성 분석에 의하여 외부 첨자 및 내부 첨자의 모든 이터레이션 사이에 종속이 존재하는 NestLoop3 클래스가 있다고 가정하자.

(그림 14)를 제안한 알고리즘에 의하여 묵시적인 병렬성을 검출하여 자바 스레드 프로그램으로 변환하면 (그림 15)와 같으며, 중첩 for 루프는 외부 및 내부 첨자에대하여 종속성이 존재하므로 외부 첨자 i에 대하여 동기화 문장 send와 wait를 삽입하여 자바의 다중스레드 프로그램으로 실행할 수 있다.

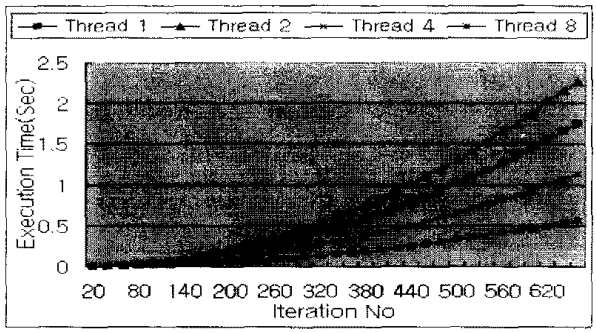


(그림 15) 그림 13의 스레드 프로그램
(Fig. 15) Thread program of figure 13



(그림 14) 예제 프로그램-3
(Fig. 14) Example program-3

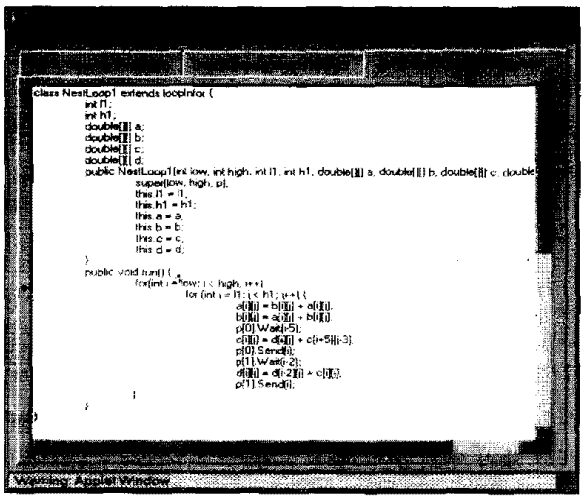
(그림 16)은 (그림 14)와 (그림 15)의 자바 프로그램에서 이터레이션 개수 N의 값 변화에 따른 실행 시간을 나타낸다. (그림 16)은 (그림 15)의 변환된 다중스레드 프로그램이 1개의 동기화 문장을 포함하므로 스레드가 2개일 때는 (그림 14)의 단일 스레드 프로그램보다 실행 시간이 느리지만 스레드가 4개 이상일 때는 실행 시간이 향상됨을 알 수 있다. 스레드가 1개인 원시 프로그램 보다 변환된 프로그램에서 스레드가 8개일 경우에 이터레이션 수가 560에서 67.4%의 성능이 향상되었다.



(그림 16) NestLoop3 클래스의 성능 분석
(Fig. 16) Performance analysis of NestLoop3 class

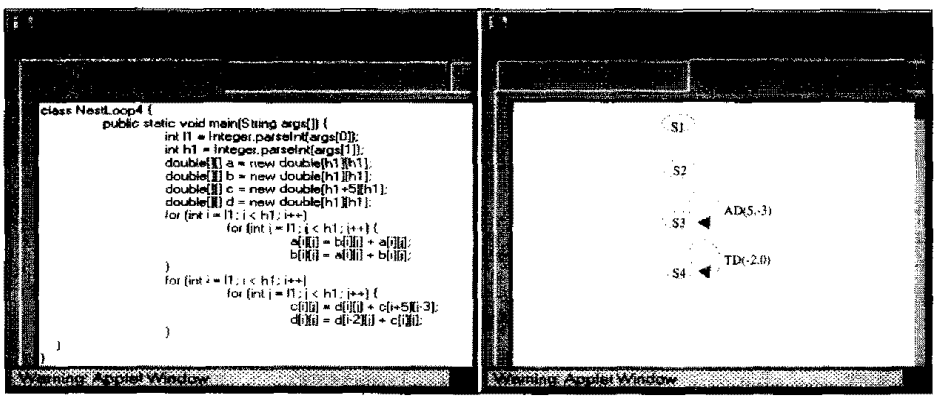
(그림 17)과 같이 2개의 중첩 for 문장 구조를 가지고 있고 종속성 분석에 의하여 첫 번째 중첩 for 문장 사이에는 종속이 존재하지 않고, 두 번째 중첩 for 문장 사이에는 종속이 존재하는 NestLoop4 클래스가 있다고 가정하자.

(그림 17)을 제안한 알고리즘에 의하여 묵시적인 병렬성을 검출하여 자바 스레드 프로그램으로 변환하면 (그림 18)과 같으며, 두 번째 중첩 for 루프는 이터레이션 사이에 종속성이 존재하므로 동기화 문장 send와 wait를 삽입하였고, 2개의 중첩 for 루프 사이에는 종속성이 존재하지 않으므로 하나의 클래스에서 실행할 수 있다.

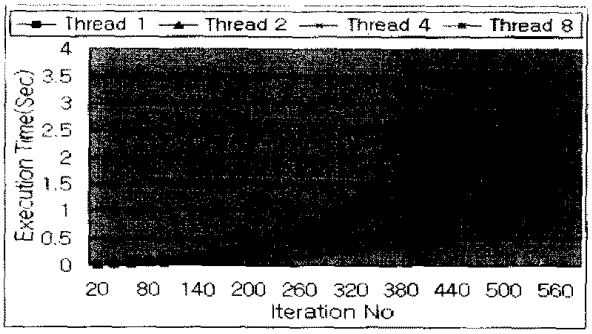


(그림 18) 그림 16의 스레드 프로그램
(Fig. 18) Thread program of figure 16

(그림 19)는 (그림 17)의 원서 프로그램과 (그림 18)의 변환된 자바 다중스레드 프로그램에서 이터레이션 개수 N의 값 변화에 따른 실행 시간을 나타낸다. (그림 18)에서 다중스레드 프로그램은 2개의 동기화 문장을 포함하므로 스레드가 2개일 때는 단일 스레드 프로그램보다 성능이 저하되고, 스레드가 4개일 때는 약간 좋아지지만, 이터레이션 수가 500에서 스레드가 8개일 때는 67.2% 정도의 성능이 향상됨을 알 수 있다.



(그림 17) 예제 프로그램-4
(Fig. 17) Example program-4



(그림 19) NestLoop4 클래스의 성능 분석
(Fig. 19) Performance analysis of NestLoop4 class

5. 결 론

순차 자바 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 자바 프로그램을 병렬 시스템에서 재사용하기 위해서는 자바 프로그램 내에서 데이터 종속성 분석을 수행한 후 묵시적인 병렬성을 찾아 자바 언어 자체에서 지원하는 다중스레드 프로그램으로 변환하는 것이 요구된다. 다중스레드는 사용자가 하드웨어의 병렬성을 추구하는데 효과적인 방법이다. 일반적으로 응용 프로그램에서 루프 구조는 전체 수행 시간 중 많은 부분을 차지하므로 병렬성 검출의 기본이 된다.

본 논문은 기존에 작성된 자바 프로그래밍 언어의 중첩 루프 구조에서 데이터 종속성 분석에 의한 종속 그래프를 구성하여 묵시적 병렬성을 검출하는 방법을 제안하였다. 또한 재구성 컴파일러에 의하여 자바 원시 프로그램을 자바 프로그래밍 언어 자체에서 지원하는 다중스레드 기법으로 변환하여 병렬 시스템에서 실행하는 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 자바 프로그램은 이터레이션 수가 증가함에 따라 원시 자바 프로그램 보다 실행 시간이 단축됨을 알 수 있었다.

본 논문에서 제안한 재구성 컴파일러에 의한 이점은 사용자의 병렬성 검출에 대한 오버헤드를 줄이고, 중첩 루프 구조를 갖는 순차 자바 프로그램에 대한 효과적인 병렬성 검출을 가능하게 하고, 병렬 시스템에서 실행 시간을 단축할 수 있었다.

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 Supercompile -rs for Supercomputers," MIT Press, 1989.

[3] Hans Zima and Barbara Chapman, "Superc -ompiler for Parallel and Vector Computer s," 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 Applica -tions," 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 Pallelism: 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.

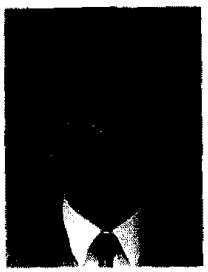
[15] James Gosling, Bill Joy, and Guy Steele, "The Java Language Specification," Addison-Wesley, Reading, Massachusetts, 1996.

[16] J. Gosling and H. McGilton, "The Java Language Environment : A white paper," Sun Microsystems, October, 1995.

[17] L. Lemay, C. L. Perkins, "Teach Yourself JAVA in 21 Days," Sams Net Pub., 1996.

[18] Sun Microsystems, "Java WorkShop 1.0," Sun Microsystems, Mountain View, California, 1996.

[19] 황득영, 정계동, 최영근, "병렬 프로그램의 사건 기반 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

최 영 근

1980년 서울대학교 사범대학 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