병렬 프로그램의 적응형 실행 기법
(Adaptive Execution Techniques for Parallel Programs)

이 재 진 †
(Jaejin Lee)

요 약 본 논문은 병렬 프로그램을 실행할 때 계산량이 작은 병렬 루프를 병렬로 실행하는 경우에 생기는 프로그램의 성능 저하를 피하기 위하여, 컴파일 시나 실행 시에 성능 예측 모델을 이용하여 병렬 루프의 성능을 예측한 다음 적응형 실행 기법을 이용하여 병렬 프로그램을 실행하는 방법을 소개한다. 성능 예측 알고리즘과 적응형 실행 알고리즘은 컴파일러 전처리기에 구현이 되었으며, 이 전처리기는 병렬 루프가 실행되는 방식을 컴파일 시나 실행 시에 결정하는 코드를 원래의 병렬 프로그램에 삽입 한다. Fortran77로 씌어진 다섯 개의 대표적인 과학 수치계산 병렬 벤치마크 프로그램을 32개의 프로세서로 구성된 분산 공유 메모리 병렬 컴퓨터(SGI Origin2000)에 실행하여 본 논문에서 제안한 방법의 성능 평가를 하였을 때, 제안한 기법을 적용한 경우가 32, 16, 8, 및 4개의 프로세서에서 원래의 병렬 프로그램 보다 각각 26%, 20%, 16%, 및 10% 빨리 실행되었다. 이중 한 프로그램은 원래 병렬 프로그램 보다 32개 프로세서에서 두 배 이상 빠르게 실행되었다.

키워드 : 병렬 프로그램 최적화, 적응형 실행 기법, 컴파일러 최적화

Abstract This paper presents adaptive execution techniques that determine whether parallelized loops are executed in parallel or sequentially in order to maximize performance. The adaptation and performance estimation algorithms are implemented in a compiler preprocessor. The preprocessor inserts code that automatically determines at compile-time or at run-time the way the parallelized loops are executed. Using a set of standard numerical applications written in Fortran77 and running them with our techniques on a distributed shared memory multiprocessor machine (SGI Origin2000), we obtain the performance of our techniques, on average, 26%, 20%, 16%, and 10% faster than the original parallel program on 32, 16, 8, and 4 processors, respectively. One of the applications runs even more than twice faster than its original parallel version on 32 processors.

Key words : Parallel Program Optimizations, Adaptive Execution Techniques, Compiler Optimizations

1. 서 론

병렬 프로그램의 성능을 높이기 위하여 고성능 최적화 병렬 컴파일러는 여러 가지 최적화 기법을 적용한다[1]. 현재 이런 컴파일러들이 당면하고 있는 최적화의 걸림돌은, 프로그램이 실행될 컴퓨터에 대한 구체적인 정보와 프로그램이 실행될 때 입력될 데이타에 대한 정보가 컴파일러에게 주어지지 않는다는 점이다. 주어진 문제를 가장 잘 풀 수 있는 알고리즘의 성능은 그것에 대한 입력 데이타와 알고리즘이 실제로 실행되는 컴퓨터에 많이 의존하며, 이러한 정보는 컴파일 할 때 얻기가 힘이 든다. 따라서 대부분의 경우, 주어진 프로그램을 컴파일 할 때 프로그램을 실행할 컴퓨터에 맞추어서 최적화하는 것은 거의 불가능하다. 이러한 문제를 해결하기 위하여 입력 데이타와 프로그램이 실행되는 컴퓨터에 대한 정보가 완전히 제공되는 시점, 즉 프로그램이 실행될 때 프로그램을 최적화하거나 프로그램이 실행 환경에 적응하도록 하여 성능을 높이는 연구가 최근에 시도되고 있다[2-6].

자동 병렬 컴파일러는 순차 프로그램을 분석하여 병렬 프로그램으로 변환한다. 하지만, 병렬 컴파일러에 의해 병렬화가 된 프로그램을 가지고 최고의 성능을 얻기 위해서는, 프로그래머가 프로그램에 들어 있는 병렬성의 양, 캐쉬 집약성(locality), 타겟 병렬 머신의 캐쉬 일관성(coherence) 기제, 프로세서 간의 계산량 및 데이타의 분배, 다중 프로세서들을 관리하기 위한 비용, 스레드 간의 동기화 비용 등을 잘 고려하여 생성된 병렬 프로

† 종신회원 : 서울대학교 컴퓨터공학부 교수
jlee@cse.snu.ac.kr
논문접수 : 2004년 6월 4일
심사완료 : 2004년 7월 12일

그램을 다시 최적화 하여야 한다. 하지만 이러한 성능 저하 요소들을 고려하여 병렬 프로그램의 성능을 높이는 일은 많은 시간이 걸리고 힘이 드는 일이다. 왜냐하면 이러한 성능 저하 요소들은 이중 하나만이 현저히 나타나는 것이 아니라 상승효과에 의해 성능에 영향을 미치며 프로그램이 실행되는 병렬 컴퓨터의 종류에 따라 다르게 나타나기 때문이다. 따라서 프로그래머가 병렬 프로그램의 적당한 성능 향상을 위해 지불해야하는 대가는 병렬 프로그램의 개발과 유지 보수에 큰 영향을 미친다.

본 논문은 병렬 프로그램의 성능 저하 요소를 프로그래머가 찾아서 없애는 대신에 프로그램이 실행 될 때, 병렬 루프의 병렬 실행에 의한 성능 저하가 미리 정의된 경계값을 넘어 가면, 그 루프가 프로그램의 실행이 끝나기 전 다시 실행될 때 루프를 순차적으로 실행 시켜 성능 저하를 피하는 적응형 실행 기법 및 비용 예측 기법을 제안한다. 적응형 실행 기법과 비용 예측 알고리즘은 컴파일러 전처리기로 구현하였고, 이 전처리기는 비용 예측 모델을 이용하여 프로그램 내의 병렬루프들이 순차적, 또는 병렬로 실행될지를 컴파일할 때 결정하거나, 그렇지 않으면 프로그램을 실행할 때에 결정하는 적응형 코드를 원래의 프로그램에 삽입한다. Fortran77로 쓰여진 대표적인 과학 수치계산 프로그램 다섯 개를 제안하는 방법으로 컴파일하고 실행하여 32개의 프로세서로 구성된 SGI Origin 2000 병렬 컴퓨터에서 26%, 20%, 16%, 및 10%의 성능 향상을 각각 32, 16, 8, 및 4개의 프로세서에서 얻었다. 이중 어떤 프로그램은 원래의 병렬 프로그램을 실행하는 것 보다 두 배 이상의 성능향상을 32개 프로세서에서 보였다.

본 논문의 나머지는 다음과 같이 구성되어 있다. 제2장은 사용한 병렬 프로그래밍 모델과 실행 모델을 기술하고 제3장은 제안한 알고리즘들을 기술한다. 제4장은 제안한 기법의 성능 평가를 위해 사용한 실험 환경을 기술하고 제5장은 제안한 기법을 사용 했을 때 나오는 성능 향상의 결과를 보인다. 제6장에서 관련된 연구를 기술하고 제7장에서 끝을 맺는다.

2. 실행 모델과 접근 방법

2.1 병렬 실행 모델

이 논문에서 사용하는 병렬 실행 모델은 주종(master-slave) 스레드(thread) 모델이다[7-9]. 한 개의 스레드가 병렬 실행 영역(parallel region)에 이르게 되면, 각자의 실행 문맥(context)를 가진 여러 개의 노예(slave) 스레드를 생성하여 팀을 만들고 이 팀의 주인(master) 스레드가 된다. 여기서 병렬 실행 영역이란 여러 개의 스레드에 의해 병렬로 실행되는 코드 블록을 일컫는다. 본 논문에서 병렬 실행 영역은 병렬 루프가 된다.

병렬 루프의 실행이 시작되면, 주인 스레드는 노예 스레드를 생성하고 루프의 반복횟수(iteration)를 팀에 속한 스레드 사이에 분배한다. 그다음 이들 스레드는 할당받은 반복횟수만큼 루프의 본체를 동시에 실행한다. 병렬 실행 영역(병렬 루프)의 끝에는 암묵적인 동기화 장벽(barrier)이 존재하는데, 자신에게 분배된 일을 다 끝낸 스레드는 다른 스레드가 일을 끝낼 때까지 여기서 기다려야 한다. 모든 스레드가 할당받은 일을 다 끝내면 동기화 장벽을 빠져 나오게 된다. 동기화 장벽을 빠져나오면 노예 스레드는 모두 소멸 되고 주인 스레드는 병렬 실행 영역 다음부터 수행을 계속한다[7-9]. 여기서 야기되는 성능에 영향을 미치는 오버헤드(overhead)는 노예 스레드를 생성하는 오버헤드, 루프내 계산량을 스레드 사이에 분배하는 오버헤드, 캐쉬 친화도(affinity)의 변화에 의한 오버헤드, 스레드 간 동기화에 의한 오버헤드 등이 있는데, 병렬 루프의 성능에 영향을 미치는 이러한 오버헤드들을 통틀어 병렬 루프 오버헤드(parallel loop overhead)라 부르겠다.

2.2 접근 방법

그림 1은 우리가 이 논문에서 문제를 풀기위해 접근하는 방법의 개요를 보여 준다. 병렬 컴파일러에 의해 생성된 병렬 프로그램이나 사람 손으로 쓰여진 병렬 프로그램이 구현된 컴파일러 전처리기에 입력으로 주어진다. 또, 타겟이 되는 병렬 컴퓨터에 대한 파라미터도 입력으로 주어지는데, 전처리기는 이를 이용하여 비용 예측 및 환경 적응 코드를 병렬 프로그램에 삽입한다. 생성된 적응형 병렬 프로그램은 다시 목표가 되는 병렬 컴퓨터를 위한 컴파일러에 의해 컴파일 되며 실행이 될 준비가 된다.

적응형 프로그램을 만들기 위해 본 논문에서 사용하는 기법은, 한 병렬 루프에 대한 여러 가지 최적화된 코드 버전 중에서 실행 환경에 따라 하나를 선택하여 실행하는 기법이다. 본 논문의 기법은 한 개의 병렬 루프에 대하여 두 개의 실행 코드 버전을 생성하는데, 그 중 하나는 순차 실행을 위한 것이고 나머지는 병렬 실행을

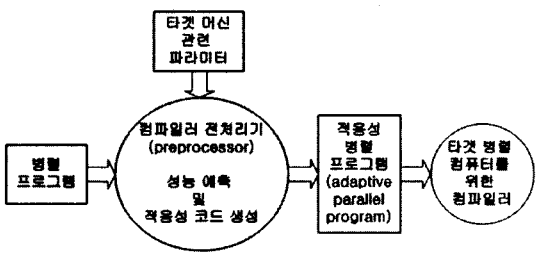

그림 1 접근 방법

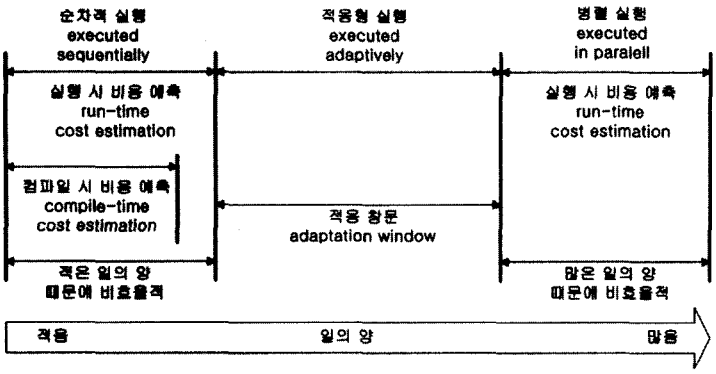

그림 2 적응형 실행 기법의 개요

위한 것이다. 이 두 버전 중 하나를 환경 적응 코드가 실행 환경에 맞게 선택하여 실행한다. 먼저, 전처리기가 각 병렬 루프 전체가 한번 실행되는 시간 또는 한번 실행되었을 때 실행되는 명령어의 개수를 세는 측정 코드를 원래의 코드에 삽입한다. 한 병렬 루프 전체의 실행 시간이나 실행된 명령어의 개수를 측정하여 루프가 실행되는 최적의 방식을 결정하는 이러한 실행 기간을 앞으로 판단 실행(decision run)이라 부르겠다. 물론 이 판단 실행도 병렬 프로그램 전체의 실행 중에 일어난다. 판단 실행에 의해 측정된 결과를 이용하여 비용 예측을 하고, 프로그램이 실행 될 동안 해당되는 병렬 루프가 다시 실행 될 때 실행되는 방법(즉, 순차 혹은 병렬)을 결정하는 환경 적응 코드가 다시 전처리기에 의해 프로그램에 삽입되며, 이와 함께 루프의 순차 및 병렬 버전에 대한 코드가 생성된다.

판단 실행 기간을 이용하는 이러한 적응형 실행 기법의 단점은 성능에 관한 유용한 정보를 얻어내기 위하여 병렬 루프를 적어도 한번은 순차적으로 또는 병렬로 실행하여야 하는 판단 실행에 있다. 즉, 병렬 루프가 적어도 한 번은 최적이 아닌 상태에서 실행된다는 것이다. 예를 들어, 순차 실행이 효율적인 루프라도 판단 실행을 위해 병렬 실행이 필요하면 적어도 한번은 병렬로 실행되어야 한다. 또, 판단 실행이 가능하려면 각 루프가 두 번 이상 프로그램이 실행될 때 실행되어야 한다는 점이다. 하지만, 실제로 병렬 처리가 많은 과학 수치계산 프로그램은 대부분의 병렬 루프가 프로그램이 한 번 실행될 때 두 번 이상 실행이 된다.

3. 적응형 실행 알고리즘

본 논문의 적응형 실행 기법은 기본적으로 컴파일 시 비용 예측 모델, 실행 시 비용 예측 모델, 및 앞의 두 비용 예측 모델을 이용한 환경 적응 기법의 세 부분으로 나뉜다. 컴파일 시 비용 예측 모델은 병렬 루프 오버헤드보다 적은 양의 계산을 하는 병렬 루프를 골라낸다(그림 2 참조). 실행 시 비용 예측 모델은 각 병렬 루프 전체가 한 번 실행될 때 마다 수행되는 명령어의 개수를 이용하여 병렬 루프 오버 헤드 보다 계산량이 많은 효과적인 병렬 루프를 골라낸다. 효과적인 병렬 루프(efficient parallel loop)란 병렬로 실행되었을 때 순차 실행에 대한 속도증가율(speedup=순차실행시간/병렬실행시간)이 1보다 큰 병렬 루프를 일컫는다. 또, 실행 시 비용 예측 모델은 컴파일 시 비용 예측 모델에 의해 비용 예측이 잘 되지 않는 계산량이 적은 병렬 루프를 골라낸다. 여러 가지 적응형 실행 알고리즘들이 이들 모델들이 골라내는 병렬 루프를 제외한 병렬 루프들의 실행 방식을 결정한다. 여기에 해당되는 루프들은 적응 창문(adaptation window) 안에 들어 있다(그림 2).

3.1 컴파일 시 비용 예측 모델

컴파일 시 비용 예측 모델은, 충분치 못한 계산량을 담고 있기 때문에 효과적으로 병렬 실행이 되지 않는 병렬 루프를 골라낸다. 이런 작은 루프를 병렬로 실행하면, 계산량이 병렬 루프 오버헤드보다 작기 때문에 순차적으로 실행하는 것보다 느리다. 따라서 이러한 루프는 병렬 실행 환경이라도 순차적으로 실행하는 것이 성능 향상에 도움이 된다. 작고 비효율적인 병렬 루프를 골라내기 위하여, 컴파일 시 비용 예측 모델은 병렬 루프에 들어 있는 계산량에 대한 경계값(threshold value)을 이용한다. 만약 병렬 루프에 들어 있는 계산량이 이 값보다 작으면 이 루프는 순차적으로 실행된다.

보통 과학 수치계산 프로그램에서 루프에 들어 있는 계산량(W)은 루프의 반복횟수(n), 루프 내의 저장 연산(assignment)의 개수(n_a), 부동소수점 덧셈 연산의 개수(n_{fadd}), 부동소수점 뺄셈 연산의 개수(n_{fsub}), 부동소수점 곱셈 연산의 개수(n_{fmul}), 부동소수점 나눗셈 연산의 개

수(n_{fdiv}), 내재 함수 호출의 개수(n_f), 및 사용자 정의 함수의 개수(n_u)에 의해 결정된다고 할 수 있다. 여기서 정수 연산의 영향은 무시할 만큼 작다. 따라서 루프의 한 반복에 들어 있는 계산량(W_i)은 다음의 식에 의해 대략적으로 예측을 할 수 있다.

$$W_i = c_a n_a + c_{fadd} n_{fadd} + c_{fsub} n_{fsub} + c_{fmul} n_{fmul} + c_{fdiv} n_{fdiv} + c_f n_f + c_u n_u$$

여기서 c_{op}는 op 종류의 연산 하나를, 타겟으로 하는 컴퓨터에서 실행할 때 드는 비용이다. 따라서 루프에 든 총 계산량은 다음과 같이 주어진다.

$$W(n) = n \cdot W_i$$

하지만 일반적으로, 루프의 반복횟수를 컴파일 시에 알 수 없으므로, n는 실행 시에 결정된다. 만약에 분기가 루프안에 들어 있다면 분기의 각 줄기에 대해 같은 비중을 두어서 계산량을 예측한다.

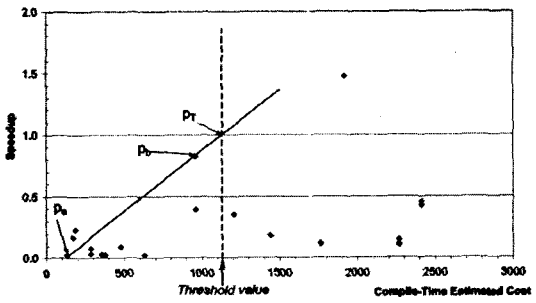

그림 3 컴파일 시 비용 예측 모델의 경계값 결정하기

앞에서 언급한 경계값(threshold value)은 휴리스틱(heuristics)을 이용해서 결정한다. 먼저, 여러 가지 다른 종류의 병렬 루프를 가진 여러 개의 벤치마크 프로그램들을 실행한다. 이 들 프로그램에 든 병렬 루프의 순차 실행 시간과 p개의 프로세서에서 실행했을 때의 병렬 실행 시간을 측정하고, 이 데이타를 이용하여 각 루프의 속도증가율(speedup)을 구한다음, 이를 Y축으로 하고, X축은 컴파일 시 비용 예측 모델에 의해 예측된 계산량으로 하여 좌표계에 점을 찍는다. 그다음 가장 작은 예측 계산량을 가진 점(그림 3에서 p_a)을 시작으로 속도증가가 0.8이상이 되는 점들 중에서 가장 작은 예측 계산량을 가진 점(그림 3에서 p_b)으로 직선을 긋는다. 경계값은 속도증가가 1.0이 되는 수평선과 이 직선이 교차하는 점(p_T)의 예측 계산량이 된다.

루프가 다중 중첩이면, 안 쪽에 있는 루프의 반복횟수가 바깥쪽에 있는 루프에 의해 결정되는 경우가 많으므로 바깥쪽의 루프를 실행하기 전에 그 값을 알아내기 어렵다. 이 경우는 실행 시 비용 예측 모델이 처리한다.

3.2 실행 시 비용 예측 모델

컴파일 시 비용 예측 모델은 작은 양의 계산을 담고 있고 비효율적인 병렬 루프를 골라내는 반면에 실행 시 비용 예측 모델은 병렬 루프 오버헤드를 상쇄할 수 있는 계산량 이상을 담고 있는 효율적인 병렬 루프를 골라낸다. 또 이 모델은 실행 시 결정되는 인자 때문에 컴파일 시 비용 예측 모델이 골라 낼 수 없는 비효율적인 병렬 루프를 골라내는 일을 한다.

실행 시 수행되는 명령어의 개수가 루프에 담겨 있는 계산량에 비례하기 때문에 병렬 루프가 실행하는 명령어의 개수를 가지고 비용 예측을 할 수 있다. 병렬 루프의 계산량이 병렬 루프를 실행하는 각 스레드에 동일하게 분배되므로, 이 경우 주인 스레드가 실행하는 명령어의 개수를 측정하여 병렬 루프내의 계산량을 예측한다. 하지만, 이러한 실행 시 비용 예측은 해당되는 병렬 루프가 프로그램이 실행 될 때 적어도 한번은 실행되어야 가능하다.

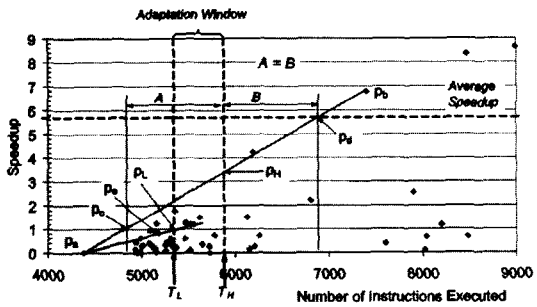

그림 4 실행 시 성능 예측 모델의 경계값 결정하기

실행 시 비용 예측 모델은 두개의 경계값을 가지고 있는데, 그 중 하나는 다른 하나 보다 작은 값으로서 실행 시 결정되는 인자 때문에 컴파일 시 비용 예측 모델이 골라 낼 수 없는, 작은 계산량을 가진 비효율적인 병렬 루프를 골라 낼 때 사용된다. 다른 하나의 값은 효율적인 병렬 루프를 골라내는 데 사용된다. 이 경계값들은 컴파일 시 비용 예측 모델의 경우와 마찬가지로 휴리스틱을 이용하여 결정한다. 여러 가지 다른 종류의 병렬 루프를 가진 여러 개의 벤치마크 프로그램들을 실행하여 이 들 프로그램에 든 병렬 루프를 순차 실행했을 때의 실행 시간과 p개의 프로세서로 병렬 실행했을 때의 실행 시간 및 실행된 명령어의 개수를 측정하고, 이 데이타를 이용하여 각 루프의 속도증가율을 구한다음 이를 Y축으로 하고, X축은 실행된 명령어의 개수로 각 루프 당 하나의 점을 좌표계에 찍는다. 그다음 가장 작은 수의 명령어가 실행된 루프에 해당되는 점(그림 4에서 p_a)에서 점이 찍혀진 모든 루프들의 속도증가율 평균값보다 큰 점들 중에서 가장 작은 수의 명령어가 실행

된 점(그림 4에서 p_b)으로 직선을 긋는다. 두개의 경계값 중 큰 값(T_H)은, 속도증가율이 1.0인 선과 이 직선의 교점(p_c)과, 평균 속도증가율을 나타내는 수평선과 이 직선의 교점(p_d)의 중간에 위치하는 점(p_H)의 실행된 명령어 개수가 된다. 경계값 중에 작은 값은 컴파일 시 비용 예측 모델의 경우와 구하는 방법이 같다. 가장 작은 예측 비용을 가진 점(그림 4에서 p_a)에서 속도증가율이 0.8이상이 되는 점들 중에서 가장 작은 예측 비용을 가진 점(p_e)으로 직선을 긋는다. 이 선과 속도증가율이 1.0이 되는 수평선과 교차하는 점(p_L)의 명령어 개수가 그 값(그림 4에서 T_L)이 된다. 작은 경계값과 큰 경계값 사이의 영역을 적응 창문(adaptation window)이라고 부른다(그림 2 참조).

3.3 적응형 실행 알고리즘

판단 실행의 횟수와 비용 예측 모델의 종류에 따라 본 논문은 다섯 개의 서로 다른 적응형 실행 알고리즘을 제안한다. 이들의 이름은 First 2 Invocations with Timing(F2T), Most Recent with Timing(MRT), Static cost estimation and MRT(SMRT), MRT with Run-time cost estimation(MRTR), 및 MRT with Static and Run-time cost estimation(SMRTR)이다. 제안하는 적응형 실행 알고리즘은 대부분의 과학 수치 계산 프로그램들이 가장 외곽의 순차 루프를 가지고 있고 이 안에 들어 있는 병렬 루프가 프로그램이 실행될 동안 여러 번 실행된다는 데 기반을 둔다.

3.3.1 First Two Invocations with Timing(F2T)

프로그램이 실행 될 때 이 프로그램에 든 병렬 루프는 대부분이 한 번 이상 실행된다. 이 들 병렬 루프가 처음으로 실행될 때 병렬로 실행하여 실행 시간을 측정한다. 이 루프가 다시 한 번 실행이 되면 순차적으로 실행하고 실행 시간을 측정한다. 측정한 두 값을 비교하여 순차 실행 시간이 작으면 루프가 다음에 실행될 때 순차로, 그렇지 않으면 병렬로 실행 한다. 한 번 실행 방식이 결정되면 그 루프에 대한 나머지 모든 실행을 결정된 방식으로 수행한다. 이 방법의 단점은 아주 효율적인 병렬 루프의 경우 적어도 한번은 순차적으로 실행해야 한다는 데에 있다. 또한, 판단 실행에서 측정한 루프의 실행 시간이 다음에 다시 그 루프가 실행 될 때의 실행 시간을 정확하게 대표한다고 할 수 없다.

3.3.2 Most Recent with Timing(MRT)

먼저 F2T와 비슷하게 병렬 루프가 처음으로 실행 될 때 병렬로 실행하여 실행 시간을 측정한다. 이 루프가 다시 한 번 실행이 되면 순차적으로 실행하고 실행 시간을 측정한다. 측정한 두 값을 비교하여 해당되는 루프가 그 다음에 실행될 때 순차 또는 병렬로 실행 될 지를 결정한다. 하지만 F2T와 달리 루프의 나머지 모든 실행을 판단 실행에서 결정된 방식으로 수행하는 것이 아니라 판단 실행의 결정을 바탕으로 다음에 루프가 실행될 때 다시 실행 시간을 측정하여 이 값을 가장 최근에 반대의 방식으로 실행했을 때 측정한 시간과 비교하여 다음에 이 루프가 실행될 방식을 결정한다.

MRT는 루프가 불릴 때마다 수행 하는 계산량이 변할 때 유용한 적응형 실행 방식이다. 즉, 가장 최근의 역사를 가지고 미래의 실행 방식을 결정하는 것이다. 따라서 루프에 든 계산량이 점진 적으로 변할 때 이 방법은 아주 효과적인 적응 방법이 된다. 하지만, 루프에 든 계산량이 루프가 실행될 때마다 급격하게 변할 때는 효과적인 방법이 될 수 없다. F2T와 마찬가지로 MRT는 아주 효율적인 병렬 루프가 적어도 한번은 처음의 판단 실행에서 순차적으로 실행이 되어야 하는 단점이 있다.

3.3.5 Static Cost Estimation and Most Recent with Timing(SMRT)

컴파일 시 비용 예측 모델과 MRT를 병행하여 병렬 루프를 비효율적으로 실행하는 것을 부분적으로 피할 수 있다. 그림 2에서 보인 바와 같이 병렬 루프 오버헤드 보다 작은 계산량을 가진 루프를 컴파일 시 비용 예측 모델을 사용하여 골라내고 순차적으로 실행하도록 하고 나머지 병렬 루프를 MRT를 사용하여 실행한다.

3.3.6 MRT with Run-time Cost Estimation(MRTR)

MRTR은 적응 창문의 개념을 사용한다. 이 적응 창문은 실행 시 비용 예측 모델의 두 경계값으로 구성된다. 병렬 루프가 처음 불려 실행 될 때 병렬로 실행되고 그것의 실행시간과 실행된 명령어의 개수를 기록한다. 만약에 실행된 명령어의 개수가 두 경계값 중에 작은 값보다 작으면 그 루프는 다음에 불릴 때 순차적으로 실행된다. 이와 반대로 실행된 명령어의 개수가 두 경계값 중에 큰 값보다 크면 다음에 병렬로 실행된다. 이 두 경우 중 어느 경우도 해당되지 않으면, 즉 실행된 명령어의 개수가 적응 창문 안에 들어 있으면, 다음 실행부터 MRT를 따라 실행이 된다. 여기서 적응 창문은 아주 효율적인 루프와 아주 비 효율적인 루프를 골라내는 역할을 한다.

3.3.7 MRT with Static and Run-Time Cost Estimation(SMRTR)

SMRTR은 컴파일 시 비용 예측 모델과 MRTR의 조합이다. MRTR이 적용되기 전에 먼저 병렬 루프 오버헤드 보다 작은 계산량을 가진 비효율적인 병렬 루프들을 컴파일 시 비용 예측 모델을 사용하여 골라내고 항상 순차적으로 실행한다. 그런 다음 나머지 병렬 루프들을 실행할 때 MRTR을 사용한다. MRTR에서 실행된 명령어의 개수를 구하기 위해 적어도 한번 병렬로 실행되어야 하는 비효율적인 병렬 루프들이 SMRTR에서

컴파일 시 비용 예측 모델에 의해 걸러지므로 판단 실행에 의한 손해를 줄일 수 있다. 결론적으로 SMRTR이 가장 좋은 성능향상을 가져다 줄 것이라 기대한다.

4. 성능 평가 환경

4.1 컴파일러

제3장에서 언급한 비용 예측 및 적응 알고리즘들은 컴파일러 전 처리기에 구현되어 있다. 이 전처리기는 Perl로 구현하였다. 순차 프로그램의 병렬화에 관한 정보는 SGI MIPSpro Fortran 77 컴파일러의 자동 병렬화 옵션(APO)을 사용하여 얻는다[10]. 병렬화에 관한 정보와 원래의 순차 프로그램이 전처리기의 입력으로 주어진다. 전처리기는 비용 예측 및 적응 코드와 타겟 병렬 컴퓨터의 컴파일러에 대한 지시문(directive)을 원래의 프로그램에 삽입한다. 이 프로그램은 다시 타겟 병렬 컴퓨터(SGI Origin 2000)의 컴파일러에 의해 병렬 프로그램으로 컴파일이 된다. 전처리기에 의해 삽입되는 SGI MIPSpro Fortran77 컴파일러의 지시문들은 표 1에 요약되어 있다. 실행시간 측정은 프로세서 사이클 카운터를 읽는SGI syssgi 라이브러리의 시스템 호출을 사용하였고, 실행된 명령어의 개수를 측정하기 위하여 SGI perfex 라이브러리를 사용하여 프로그램 실행 중에 프로세서의 이벤트 카운터를 읽었다.

4.2 벤치마크 프로그램

앞에서 제안한 알고리즘들의 성능 평가에 Fortran77로 쓰여진 대표적인 다섯 개의 과학기술 수치계산 벤치마크 프로그램을 사용하였다. 표 2는 이들 응용 프로그램의 이름과 크기 및 사용된 입력 데이타 크기, 반복횟수를 나타낸다.

4.3 타겟 병렬 컴퓨터

제안한 시스템에 의해 생성된 코드는 SGI Origin 2000에 의해 실행된다[11]. 모든 실험은 SGI Origin 2000를 다른 응용 프로그램의 실행이 없는 상태에서 수행되었다. 표 3은 사용한 SGI Origin2000의 특성을 나타낸다.

5. 성능 평가 및 결과

5.1 병렬 루프의 특성

표 4는 각 응용 프로그램에 든 병렬 루프의 특성을 요약한 것이다. 표에 두개의 구역이 있는데, 그 중 하나는 성능 평가에 사용한 프로그램에 든 모든 병렬 루프에 관한 특성이고 다른 구역은 그중 비효율적인 병렬 루프에 관한 특성이다. 첫 구역의 첫 줄은 각 응용 프로그램에 든 병렬 루프의 총 수와 그 루프가 프로그램의 순차 실행 시간에서 차지하는 비율을 %로 나타낸 것이다. 그 다음 줄은 각 병렬 루프가 응용 프로그램이 실행 될 때 실행되는 평균 실행 횟수를 보여 준다. 마지막 줄은 각 병렬 루프가 한번 실행 될 때 걸리는 평균시간을 프로세서 사이클로 표현한 것이다. 이것이 병렬 루프들의 서로 다른 프로세서 개수에 대한 평균 크기를 나타 낸다.

표 1 SGI MIPSpro Fortran 77 컴파일러에 사용되는 자동 병렬화 지시문들

Directive	의미
C*$* ASSERT DO (SERIAL)	바로 다음에 나오는 루프를 병렬화 하지 않는다.
C*$* ASSERT DO PREFER (CONCURRENT)	바로 다음에 나오는 루프를 병렬화 한다.

표 2 사용된 벤치마크 프로그램들

응용 프로그램	출처	라인 수	데이타 크기 및 반복횟수(iteration)
Applu	SPECfp2000	3980	Reference input, 20 iteration
Hydro2d	SPECfp95	4303	Reference input, 100 iteration
Mgrid	SPECfp2000	489	Test input, 40 iteration
Su2책	SPECfp95	2271	Reference input, 100 iteration
Swim	SPECfp2000	435	Reference input, 50 iteration

표 3 성능 평가에 사용된 SGI Origin2000의 파라미터

Architecture type	Distributed Shared Memory
프로세서의 종류 (clock)	MIPS R10000 (250Mhz)
프로세서의 수	128
총 메모리 크기	128GB
총 디스크 크기	640GB
명령어 캐쉬 크기 (캐쉬 라인 크기)	32KB(64B)
데이타 캐쉬 크기 (캐쉬 라인 크기)	32KB(32B)
2 번째 명령어/데이타 병합 캐쉬 크기 (캐쉬 라인 크기)	4MB(128B)

표 4 성능 평가에 사용한 프로그램에 든 병렬 루프의 특성

			Applu	Hydro2d	Mgrid	Su2cor	Swim	평균
병렬 루프	병렬 루프의 수 (순차 실행 시간 %)		55 (95.2%)	86 (97.3%)	11 (99.7%)	41 (88.8%)	16 (99.8%)	41.8 (96.2%)
	평균 실행 횟수		8779.7	373.0	422.7	34220.1	28.4	8764.8
	평균 루프 크기 (프로세서 사이클)		127.3K	4.1K	44.3K	0.6K	364.6K	108.2K
비효율적인 루프	비효율적인 루프의 수 (순차 실행 시간 %) (병렬 실행 시간 %)	# procs	Applu	Hydro2d	Mgrid	Su2cor	Swim	평균
		2	28 (0.9%) (16.5%)	34 (0.6%) (13.3%)	1 (0.5%) (0.9%)	15 (2.8%) (25.2%)	3 (0.9%) (10.2%)	16.2 (1.2%) (13.2%)
		4	29 (0.9%) (28.1%)	28 (0.2%) (20.8%)	-	15 (2.8%) (34.5%)	3 (4.2%) (21.6%)	18.8 (2.0%) (26.3%)
		8	28 (0.9%) (44.3%)	28 (0.2%) (28.9%)	-	16 (2.8%) (48.6%)	3 (4.2%) (33.4%)	18.8 (2.0%) (38.8%)
		16	29 (0.9%) (57.2%)	28 (0.2%) (37.1%)	1 (0.2%) (1.2%)	18 (2.8%) (62.5%)	3 (4.2%) (51.8%)	15.8 (1.7%) (42.0%)
		32	32 (1.4%) (79.6%)	32 (0.3%) (44.8%)	-	18 (2.8%) (70.0%)	6 (4.2%) (67.1%)	22.0 (2.2%) (65.3%)
	평균 실행 횟수	2	16871.3	357.7	1042.0	80892.9	17.3	19836.2
		4	16289.6	309.4	-	80892.9	2.3	24373.6
		8	16871.3	337.9	-	75837.2	2.3	23262.2
		16	16289.6	312.9	1.0	67458.8	2.3	16812.9
		32	15077.8	351.9	-	67458.8	18.0	20726.6
	평균 루프 크기 (프로세서 사이클)	2	1.4K	0.6K	0.4K	0.2K	368.7K	74.3K
		4	3.7K	0.6K	-	0.2K	633.4K	159.5K
		8	1.4K	0.5K	-	0.5K	633.4K	159.0K
		16	1.4K	0.5K	179.2K	0.4K	633.4K	163.0K
		32	1.3K	0.4K	-	0.4K	317.0K	79.8K

또 다른 구역은 다시 세 영역으로 나뉘는데, 그 첫 영역은 2, 4, 8, 16, 및 32개의 프로세서에 대하여 각 프로그램에 들어 있는 비효율적인 병렬 루프의 총 수를 보여 준다. 또, 이들 루프의 순차 및 병렬 실행 시간이 프로그램 전체의 순차 실행 시간 및 병렬 실행 시간에서 차지하는 비율을 %로 각각 보여 준다. 그 다음 영역은 비효율적인 병렬 루프들이 프로그램이 실행될 때 실행되는 평균 횟수를 2, 4, 8, 16, 및 32개의 프로세서에 대하여 보여 준다. 마지막 줄은 2, 4, 8, 16, 및 32개의 프로세서에 대하여 이들 루프가 한 번 실행될 때의 평균 병렬 실행 시간을 프로세서 사이클로 표현한 것이다. 이는 비효율적인 병렬 루프들의 서로 다른 프로세서 개수에 대한 평균 크기를 나타낸다.

표에서 볼 수 있듯이 사용된 프로그램들은 병렬성을 아주 많이 포함하고 있으며 평균적으로 병렬 루프의 순차 실행시간이 순차 실행 시간의 96.2%를 차지한다. 특히 Applu, Mgrid, Hydro2d, 및 Su2cor은 많은 수의 병렬 루프들을 포함하고 있다. 이중 30% 이상의 병렬 루프가 비효율적인 루프이다. 하지만 이들 비효율적인 루프는 순차 실행 시간에 별 영향을 미치지 않는데 평균적으로 약 2%의 순차 실행 시간을 점유할 뿐이다. 이와 반대로 Mgrid를 제외한 모든 프로그램에서 이들 비효율적 루프는 병렬 실행 시간을 장악하고 있다. 2개의 프로세서에서 이들 비효율적 루프들은 13.2%의 병렬 실행 시간을 차지하며 32개의 프로세서에서 65.3%의 병렬 실행 시간을 차지한다. 따라서 이들 비효율적인 루프가 병렬 프로그램 최적화의 좋은 목표가 됨을 알 수 있다.

또, 대부분의 병렬 루프에 대하여 그 실행 횟수가 많음을(평균 8764.8회) 알 수 있으며 비효율적인 병렬 루프가 그렇지 않은 다른 병렬 루프에 비해 더 많이(예를 들어 32 프로세서에서 20726.6회) 실행됨을 알 수 있다. Mgrid와 Swim을 제외하고 이들 비효율적인 루프의 크기(프로세서 사이클)는 그렇지 않은 병렬 루프에 비해 훨씬 작다. 따라서 비효율적인 병렬 루프를 병렬로 실행했을 때 일어나는 오버헤드는 실제 루프 자체의 실행 시간보다 매우 큼을 알 수 있다. 이 사실은 우리가 제안

하는 기법이 병렬 프로그램에 대해 효과적인 최적화 기법임을 보여 준다.

5.2 성능 평가

그림 5는 3장에서 제안한 여러 가지 기법들에 대한 각 벤치마크 프로그램들의 실행 시간이다. 이 그림은 각 프로그램에 대하여 제안한 기법들을 가지고 2, 4, 8, 16, 및 32개 프로세서에서 실행한 실험결과 및 그 평균을 보여 준다. 가장 왼쪽의 막대는 원래 병렬 프로그램의 병렬 실행 시간(Base)을 나타내며, 그 다음 여섯 개의 막대는 각각 제3장에서 언급한 기법인 F2T, MRT, SMRT, MRTR, SMRTR의 실행 시간에 해당 하며 Static과 Runtime은 각각 컴파일 시 비용 예측과 실행 시 비용 예측만 각각 적용하여 실행 했을 경우를 나타낸다. 이들 실행 시간은 Base에 대하여 정규화 되어 있다.

실험결과를 보면 F2T가 가장 성능이 나쁘다. 이는 병렬 루프의 처음 두 번 실행만 판단 실행으로 사용하고 다음부터 프로그램의 실행이 끝날 때까지 그 루프가 실행이 되는 방법에 변화가 없기 때문이다. 이는 병렬 루프에 들어 있는 계산량이 루프가 실행될 때마다 변하면 판단 실행에서 결정된 실행 방법이 효과적인 실행 방법이 아님을 말한다. 또, F2T는 각 병렬 루프를 판단 실행에서 적어도 한번은 최적이 아닌 방법으로 실행을 하게 된다(병렬로 한번, 순차적으로 한번 실행을 하므로).

MRT는 F2T보다 성능이 좋으며 가끔은 Base 보다 좋다. 그 이유는 각각의 루프를 적응형으로 실행을 하기 때문이다. 하지만 적응형으로 실행을 하는 과정에서 적어도 한번은 최적이 아닌 방법으로 루프를 실행하게 된다. 만약에 루프에 든 계산량이 루프가 실행될 때마다 변하면 MRT의 성능 예측 결과가 틀릴 경우가 많고 성능이 저하된다.

Static은 컴파일 시 비용 예측 모델의 예측 결과에 따라 병렬 루프를 실행한다. 그림 5에서 보는 바와 같이 Static은 대부분의 프로그램(Applu, Hydro2d, Su2cor)에서 Base보다 성능이 좋다. 그 이유는 이들 프로그램에서 Static이 작은 병렬 루프 안에 든 계산량을 잘 예측하여(다중으로 중첩된 루프는 예측을 하지 않는다) 골라내고, 또 이들 루프가 프로그램이 실행될 때 자주 실행되기 때문이다. 더욱이 컴파일할 때 비용 예측을 하기 때문에 판단 실행에 의한 비용이 들지 않는다. 표 5는 각 프로그램에 대하여 Static이 골라내는 비효율적인 병

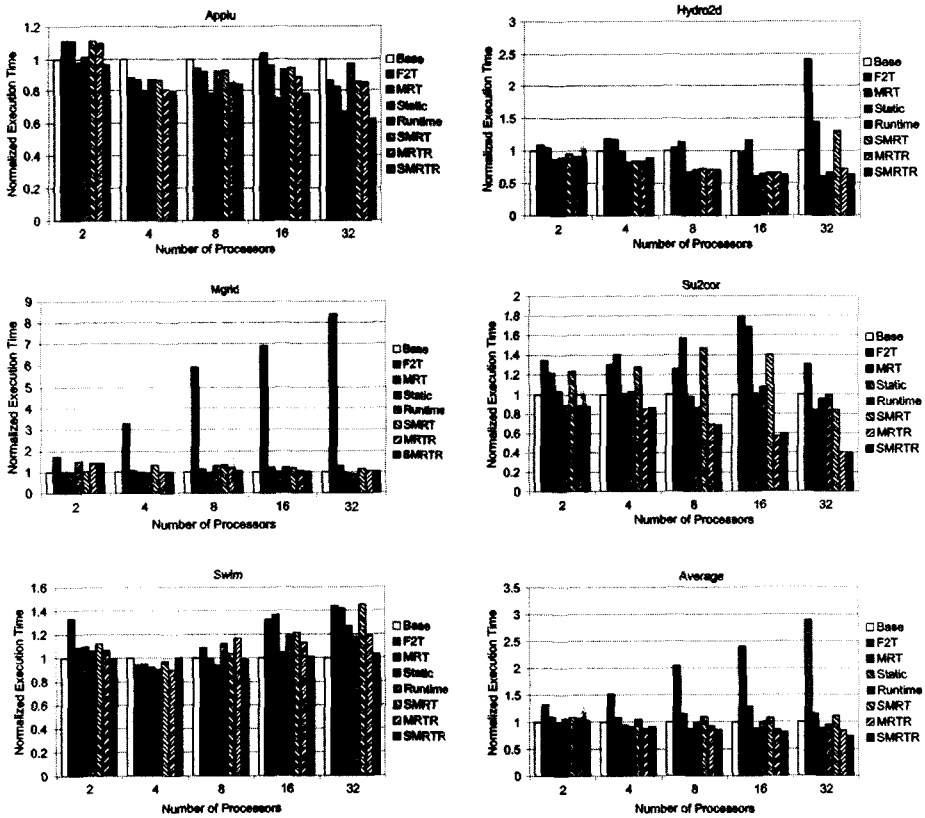

그림 5 각 프로그램에의 제안된 기법들에 대한 정규화된 실행 시간

표 5 컴파일 시 비용 예측 모델에 의해 계산량을 예측할 수 있는 비효율적 병렬 루프의 특성

프로그램	Applu	Hydro2d	Mgrid	Su2cor	Swim	평균
병렬 루프의 수 (순차 실행 시간 %)	18 (0.51%)	31 (0.36%)	1 (0.00%)	32 (88.50%)	8 (0.44%)	18 (17.97%)
평균 실행 횟수	25122.4	389.0	1.0	43801.6	37.3	13870.3

렬 루프의 개수와 그 루프들이 차지하는, 프로그램의 순차 실행시간에 대한 루프의 평균 순차 실행 시간의 비율 및 프로그램이 실행 될 때 루프가 실행되는 평균 횟수를 보여준다. Static은 방법 자체가 간단하고 웬만큼 성능 향상이 되기 때문에 좋은 최적화 기법이 될 수 있다.

Runtime은 F2T, MRT, 및 Base 보다 성능이 좋지만 Static 보다 성능이 좋지 않다. 그 이유는 루프가 실행 될 때 실행된 명령어의 개수로 비용 예측을 하므로 예측 결과는 Static 보다 정확할지 모르나, 명령어 수를 세기 위하여 비효율적인 루프라 하더라도 한번은 병렬로 실행해야 하기 때문에 성능이 많이 좋아지지 않기 때문이다.

SMRT는 F2T와 MRT 보다 성능이 좋고, Base 보다 좋지 않다. 이는 MRT의 판단 실행에서 오는 비용이 Static을 사용 했을 때 오는 성능의 이득 보다 크기 때문이다. Static에 의해 비효율적인 작은 루프가 많이 걸러지더라도 남아 있는 병렬 루프들(계산량이 아주 많은 효율적인 루프들을 포함하여)을 MRT의 판단 실행에 의해 실행 방법을 결정하기 때문에 이 비용이 성능에 많은 영향을 미치고 Base보다 성능이 좋지 않게 된다.

MRTR은 Applu, Hydro2d, 및 Su2cor에서 Base 보다 성능이 훨씬 좋다. Mgrid와 Swim의 경우는 Static과 비교할 만 하다. 그 이유는 MRTR이 실행 시의 정보를 이용하여 병렬 루프의 성능 예측을 Static 보다 더 정확하게 할 수 있고, 따라서 더 많은 수의 비효율적인 작은 병렬 루프들이 순차적으로 실행되기 때문이다. 결론적으로, 일의 양이 많은 효율적인 병렬 루프를 실행시 비용 예측 모델이 걸러 주기 때문에 MRT를 적용할 때 이들 효율적인 루프를 적어도 한 번 순차적으로 실행해야 할 일이 없다. 따라서 성능 향상에 많은 도움을 준다.

성능 평과의 결과를 보면 예상한 대로 SMRTR의 성능이 가장 좋다. 그 이유는 SMRTR이 적응 창문 안에 들어오는 루프만 판단 실행을 적용하여 실행하므로 이에 의한 성능 저하가 다른 방법보다 적기 때문이다. Mgrid와 Swim에서 판단 실행에 의한 성능 저하가 조금 있지만 MRT와 비교할 때, 판단 실행에 의한 성능 저하를 현저히 감소시킴을 알 수 있다. 결론적으로 SMRTR은 비효율적인 병렬 루프를 더욱 정확하게 골라낸다. 따라서 제안한 방법 중 SMRTR이 가장 좋은 적응형 실행 기법이라고 할 수 있다. SMRTR을 이용하

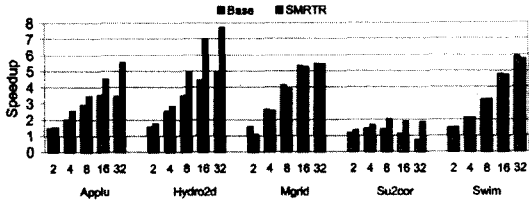

그림 6 각 프로그램에 대한 Base와 SMRTR의 속도증가율(speedup)

여 각 프로그램에서 달성한 속도증가율(speedup)은 그림 6에 나와 있다. SMRTR의 속도증가율은 대부분의 경우 원래의 프로그램을 병렬로 실행한는 Base보다 크거나 그렇지 않으면 거의 같다. Su2cor의 경우 32 프로세서에서 속도 증가가 Base보다 두 배 이상이다.

6. 관련 연구

실행 환경에 대한 완전한 정보가 알려지는 프로그램 실행 시간에 적용하는 최적화에 관한 연구가 최근 많이 수행되고 있고, 여러 가지 적응형 최적화 기법들이 연구 문헌에 나타나고 있다. 한 개의 루프에 대하여 컴파일 시 생성된 여러 개의 코드 버전중에 실행시의 피드백(feedback)을 이용하여 가장 좋은 버전 하나를 선택하여 실행하는 것은 동적 적응형 최적화의 좋은 예이다[3,12]. 이런 다중 버전 방식의 주된 문제는, 각각의 최적화 기법에 대하여 다른 코드 버전을 생성하여야 하므로 코드 크기가 매우 커진다는 것이다. 본 논문에서 제안한 방법은 한 병렬 루프에 대하여 두 개의 코드 버전(순차 및 병렬 버전)을 사용한다. 따라서 코드 크기는 많아야 두 배 정도로 커지고, 이것은 흔히 쓰이는 병렬화 컴파일러에 구현된 있는 조건부 병렬화 지시문을 사용하면 피할 수 있다.

또 실행 시에 코드 변환을 하기 위해 컴파일 시에 코드를 인자화하는 연구도 있다[13-16]. 즉 실행시의 정보를 이용하여 이 인자들을 채워 최적화를 실행시에 마무리 한다. Gupta와 Bodik은 루프 병합, 루프 분열, 루프 교환등과 같은 루프 변환을 프로그램 실행 시에 수행하는 경우의 속도를 연구하였다[14]. 객체 지향 프로그램에서 객체의 복제를 이용한 적응형 최적화 기법이 Rinard 등에 의해 제안 되었다[15]. 공유되는 객체를 수정할 때 생기는 동기화 오버헤드를 줄이기 위하여 실행

시에 그 객체를 복제하는 방법이다. 복제하는 정책은 프로그램이 실행 될 때의 환경 특성에 따라 적응형으로 적용된다. Holzle 등은 객체 지향 프로그램의 성능을 높이기 위하여 동적 타이프 피드백(type feedback) 기법을 제안하였다[17]. 이들의 방법은 프로그램이 동적으로 실행 환경에 적응한다는 측면에서 본 논문에서 제안한 방법과 비슷하지만, 본 논문의 접근 방법은 프로그램을 실행 시에 변환하여 최적화하지는 않으며, 병렬 프로그램에 적응형 실행 기법을 적용하여 성능을 높였다.

Voss와 Eigenman에 의하여 적응형 프로그램 최적화를 위한 ADAPT라는 컴파일러 지원 기반이 제안되었다[5,6]. 여기서 프로그래머는, 프로그램 실행 시에 최적화 기법을 적용하기 위하여 최적화의 종류와 휴리스틱을 ADAPT 언어를 이용하여 기술한다. ADAPT 컴파일러는 이를 이용하여 실행시 최적화를 하는 실행시간 시스템을 주어진 프로그램에 대하여 생성해 낸다. Lee는 휴리스틱과 컴파일 시의 성능 예측 방법을 통하여 비효율적 병렬 루프의 순차 실행 기법을 제안하였다[18]. 본 논문에서 제안한 기법은 [18]에서 사용한 컴파일 시의 성능 예측 모델을 사용하고 있으며 Processing-In-Memory 시스템에 대한 적응형 컴파일러 기법과도 관련이 있다[19].

7. 결 론

본 논문은 병렬 프로그램 안에 든 병렬 루프를 루프 안에 든 계산량 및 실행 환경에 따라 병렬 혹은 순차적으로 실행시켜서 성능을 향상시키는 비용 예측 방법과 적응형 실행 방법을 제안한다. 적응 알고리즘과 비용 예측 알고리즘은 원래의 병렬 프로그램에 컴파일러 전처리기가 삽입 한다. 다섯 개의 대표적인 과학 수치계산 병렬 프로그램을 사용한 성능 평가에서, 제안된 알고리즘이 32, 16, 8, 및 4 개의 프로세서에서 각각 26%, 20%, 16%, 및 10%의 성능향상을 가져오는 것을 알 수 있다. 어떤 프로그램은 32개 프로세서에서 원래 병렬 실행 결과 보다 두 배 이상 빨라진다. 이러한 결과를 볼 때, 본 논문에서 제안한 기법이 이미 병렬화 된 프로그램을 최적화하는 효과적인 방법 중의 하나라는 것을 알 수 있다.

참 고 문 헌

[1] William Blume, Ramon Doallo, Rudolf Eigenmann, John Grout, Jay Hoeflinger, Thomas Lawrence, Jaejin Lee, David Padua, Yunheung Paek, Bill Pottenger, Lawrence Rauchwerger, and Peng Tu. Parallel programming with Polaris. *IEEE Computer*, 29(12):78-82, December 1996.

[2] Bowen Alpern et al. The Jalapeno Virtual Machine. *IBM Systems journal*, 39(1):211-238, February 2000.

[3] Pedro Diniz and Martin Rinard. Dynamic Feedback: An Effective Technique for Adaptive Computing. In *Proceedings of the ACM SIGPLAN Conference on Program Language Design and Implementation*, pages 71-84, June 1997.

[4] Martin Rinard and Pedro Diniz. Eliminating Synchronization Bottlenecks in Object Based Programs Using Adaptive Replication. In *Proceedings of the ACM International Conference on Supercomputing (ICS)*, pages 83-92, June 1999.

[5] Michael J. Voss and Rodolf Eigenmann. ADAPT: Automated De-Coupled Adaptive Program Transformation. In *Proceedings of the International Conference on Parallel Processing (ICPP)*, August 2000.

[6] Michael J. Voss and Rudolf Eigenmann. High-level Adaptive Program Optimization with ADAPT. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 93-102, June 2001.

[7] OpenMP Standard Board. *OpenMP Fortran Interpretations*, April 1999. Version 1.0.

[8] R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, and R. Manon. *Parallel Programming in OpenMP*. AMorgan Kaufmann Publisher, 2001.

[9] Silicon Graphics Inc. *MIPSpro Auto-Parallelization Option Programmer's Guide.*

[10] Silicon Graphics Inc. *MIPSpro Fortran 77 Programmer's Guide.*

[11] The National Center for Supercomputing Applications. *http://www.ncsa.uiuc.edu.*

[12] Mark Byler, James Davies, Christopher Huson, Bruce Leasure, and Michael Wolfe. Multiple Version loops. In *Proceedings of the International Conference on Parallel Processing (ICPP)*, pages 312-318, August 1987.

[13] Alan L. Cox and Robert. J. Fowler. Adaptive Cache Coherency for Detecting Migratory Shared Data. In *Proceedings of the 20th International Symposium on Computer Architecture*, pages 98-108, May 1993.

[14] Rajiv Gupta and Rastislav Bodik. Adaptive Loop Transformations for Scientific Programs. In *Proceedings of the IEEE Symposium on Parallel and Distributed Processing*, pages 368-375, October 1995.

[15] Theodore H. Romer, Dennis Lee, Brian N. Bershad, and Bradley Chen. Dynamic Page Mapping Policies for Cache Conflict Resolution on Standard Hardware. In *Proceedings of the 1st USENIX Symposium on Operating Systems Design and Implementation*, pages 255-266, November 1994.

[16] Rafael H. Saavedra and Daeyeon Park. Improving

the Effectiveness of Software Prefetching with Adaptive Execution. In *Proceedings of the Conference on Parallel Algorithms and Compilation Technique*, October 1996.

[17] Urs Holzle and David Ungar. Optimizing Dynamically-Dispatched Calls with Run-Time Type Feedback. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 326-336, June 1994.

[18] Jaejin Lee. *Compilation Techniques for Explicitly Parallel Programs.* PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, October 1999. Department of Computer Science Technical Report UIUCDCS-R-99-2112.

[19] Jaejin Lee, Yan Solihin, and Josep Torrellas. Automatically Mapping Code in an Intelligent Memory Architecture. In *Proceedings of the 7th International Symposium on High Performance Computer Architecture (HPCA)*, pages 121-132, January 2001.

이 재 진

1991년 서울대학교 물리학과, 학사. 1995년 미국 Stanford University, Computer Science, 석사. 1999년 미국 University of Illinois at Urbana-Champaign, Computer Science, 박사. 2000년~2002년 미국 Michigan State University, Department of Computer Science and Engineering, 조교수 2002년~현재 서울대학교 컴퓨터공학부 조교수. 관심분야는 Compilers, Programming Languages, High Performance Computing Systems, Embedded Systems