論文2001-38CI-5-2

중앙 큐 기반의 루프 스케줄링 기법의 설계 및 구현

(Design and Implementation of the Central Queue Based Loop Scheduling Method)

金 炫 澈 *, 金 孝 喆 **, 柳 基 永 ***

(Hyun Chul Kim, Hyo Cheol Kim, and Kee Young Yoo)

요 약

본 논문에서는 루프의 반복들간에 종속 관계가 존재하는 루프의 효율적 수행을 위한 중앙 큐 기반의 새로운 할당 기법 CDSS(Carried-Dependence Self-Scheduling)를 제안하며, 이를 공유 메모리 환경에서 Java 언어로 구현하였다. 또한, 중앙 작업 큐 기반의 병렬 루프를 위한 셀프 스케줄링(self-scheduling) 기법들을 루프 캐리 종속성(loop-carried dependence)을 가진 루프의 할당에 적용하기 위한 그들의 변형에 대해 알아본다. 제안된 기법은 종속 거리에 따른 동기화 시점을 고려하여 루프를 세 단계별로 할당하는 셀프 스케줄링 기법이다. 단일처리기 시스템을 포함한 여러 플랫폼에 적용하기 위해 제안된 방법과 변형된 기법들을 스레드 레벨로 구현하였다. 응용 프로그램과 시스템 파라메터 값을 다양하게 하여 변형된 기법들과 비교 분석한 결과, 제안된 기법은 변형된 다른 기법들에 비해 스케줄링 오버헤드를 포함한 전체 루프의 수행 시간을 줄여 효율적이다. 변형된 SS, Factoring, GSS, CSS에 대해 각각 0.02, 40.5, 46.1, 53.6%의 성능 향상을 보였다. 그리고, CDSS 기법으로 다양한 응용 프로그램에 대해 종속 거리에 해당하는 적은 수의 스레드를 사용하여 최대의 성능을 얻을 수 있다.

Abstract

In this paper, we present a new scheduling method called CDSS(Carried-Dependence Self-Scheduling) for efficiently execution of the loop with intra-dependency between iterations based on the central queue. We also implemented it on shared memory system using Java language. Also, we study the modification that converts the existing self-scheduling method based on the central task queue for parallel loops onto the same form applied to loop with loop-carried dependences. The proposed method is self-scheduling and assigns the loops in three-level considering the synchronization point according to the dependence distance of the loops. To adapt the proposed scheme and modified methods into various platforms, including a uni-processor system, we use threads for implementation. Compared to other assignment algorithms with various changes of application and system parameters, CDSS is found to be more efficient than other methods in overall execution time including scheduling overheads. CDSS shows improved performance over modified SS, Factoring, GSS and CSS by about 0.02, 40.5, 46.1 and 53.6%, respectively. In CDSS, we achieve the best performance on varying application programs using a few threads, which equal the dependence distance.

* 正會員, 浦項 1大學 情報通信科
(Prof. of Dept. Information & Telecommunication, Pohang College)
** 正會員, 啓明文化大學 電算科
(Prof. of Dept. Computer Science, Keimyung College)
*** 正會員, 慶北大學校 컴퓨터工學科
(Prof. of Dept. Computer Engineering, Kyungpook Natl. University)
接受日字:2001年1月29日, 수정완료일:2001年6月12日

Ⅰ. 서 론

응용 프로그램에서 가장 많은 병렬성을 제공하는 대상으로 알려진 루프의 효율적인 스케줄링에 관한 많은 연구가 이루어지고 있다. 루프의 병렬 수행을 위해 반복들을 실행 시간에 다중처리기에 할당하는 루프의 동적 스케줄링(dynamic scheduling)기법은 사용된 작업 큐에 따라 중앙과 지역 작업 큐 혹은, 분산 큐 기반의 할당 기법으로 나누어지며, 전용 스케줄러의 존재 여부에 따라 중앙 집중식과 분산 스케줄러 혹은, 셀프 스케줄링(self-scheduling) 기법으로 구분된다[1,2,3]. 셀프 스케줄링에서는 프로세서가 수행 코드 앞, 뒤 부분에 스케줄링을 위해 추가되어진 코드 부분도 수행함으로써, 별도의 전역 스케줄러가 필요하지 않으며 스케줄러가 분산된 개념을 가진다. 즉, 프로세서가 코드 실행과 스케줄링 기능을 병행하는 것이다[1,2,3]. 이러한, 셀프 스케줄링 기법들 중에 중앙 작업 큐를 이용하여 독립적인 반복을 가진 병렬 루프를 공유 메모리 다중처리기에 할당하는 기법으로는 SS(Self-Scheduling), CSS(Chunk Self-Scheduling), GSS(Guided Self-Scheduling), Factoring, TSS(Trapezoid Self-Scheduling)등이 있으며, 지역 작업 큐를 사용하는 것으로 친화 스케줄링(Affinity Scheduling), Tapering, SSS(Safe Self-Scheduling)등이 있다[1,2,3]. 위의 스케줄링 기법들은 반복들간에 종속 관계가 존재하지 않는 병렬 루프를 위한 스케줄링 기법들이다. 루프 반복들간에 의존성이 생기는 루프 캐리 종속성(loop-carried dependence)을 가진 루프의 병렬 수행을 위해서는 종속성 만족을 위한 동기화가 필요하다[1,2,3]. 지금까지 대부분의 셀프 스케줄링에 관한 연구는 반복들 사이에 의존성이 없는 병렬 루프를 위한 것들이며, 동기화가 필요한 루프를 위한 효율적인 셀프 스케줄링 기법은 소개되지 않고 있다.

본 논문에서는 루프의 반복들 간에 종속 관계가 존재하는 루프의 효율적 수행을 위한 새로운 할당 기법을 제안한다. 그리고, 중앙 작업 큐 기반의 셀프 스케줄링 기법들 중에 SS, CSS, GSS, Factoring의 네 가지 기법들을 루프 캐리 종속성을 가진 루프의 할당에 적용하기 위한 그들의 변형에 대해 알아본다. 본 논문에서 제안한 기법 또한, 중앙 큐를 기반으로 한 것이며 별도의 스케줄러가 필요 없는 셀프 스케줄링 기법이다. 제안한 할당 기법은 종속 거리(dependence distance)에 따른 종속성 충족을 위한 동기화 시점을 고려하여 루프를 크게 세 단계별로 할당한다. 셀프 스케줄링 기법들을 단일처리기 시스템을 비롯한, 다양한 플랫폼에서 수행하기 위해 Java 언어를 사용하여 프로세서의 기능을 스레드로 구현하였다. 성능 평가를 위해 응용 프로그램의 파라메터인 종속 거리(d), 루프 반복 수(n)와 각 반복의 실행 비용(e)과 시스템 파라메터인 스레드 수(t)를 다양하게 하여 변형된 할당 정책들과 제안한 기법의 스케줄링 오버헤드를 포함한 루프의 전체 수행 시간을 비교 분석한다.

Ⅱ. 관련 연구

동적 루프 스케줄링 기법은 사용된 작업 큐와 전용 스케줄러의 존재 여부에 따라 세분화된다. 그 중, 공유 메모리 다중처리기 환경에서 프로세서 스스로가 스케줄링 기능을 행하는 셀프 스케줄링의 대표적인 기법들에 대해 알아본다. 먼저, 중앙 큐 기반의 SS 할당 기법에서는 병렬 루프의 모든 반복이 수행되어질 때까지 휴면 프로세서들이 작업 큐로부터 하나의 반복만을 가져와 수행한다[4,5]. 이 기법은 거의 완벽한 부하 균형을 유지하는 반면에, 각 프로세서가 임계 구역인 중앙 작업 큐에 반복 개수 만큼인 n번의 접근을 하기에 많은 스케줄링 오버헤드를 발생시킨다. 이를 해결하기 위해 소개된 CSS 기법은 한 번에 k개의 반복을 가져옴으로 스케줄링 오버헤드를 줄이지만, 부하 균형은 SS보다 좋지 못하다[5,6]. 그리고, 적당한 k 값을 선택하기가 힘들다. 일반적으로, k 는 $\lceil n/p \rceil$ 로 결정된다[5,6]. 여기서, p는 프로세서의 수이다. GSS 할당 기법은 루프의 시작 부분에서는 덩어리(chunk) 크기를 크게 하여 할당함으로써 스케줄링 오버헤드를 감소시키며, 루프의 끝 부분에서는 덩어리 크기를 작게 하여 부하 균형을 좋게 한다. 단점으로는, 루프 끝 부분을 수행하면서 작업 큐의 경쟁을 유발한다. GSS에서의 덩어리 크기 결정은 다음과 같이 이루어진다[7].

$$R_o = n,\ R_{i+1} = R_i - G_i$$
$$G_i = \lceil R_i/p \rceil \quad (1)$$

여기서, R_i는 i 번째 스케줄링 단계의 남은 반복의 수이며, G_i는 i 번째 스케줄링에 할당되는 덩어리의 크기이다. 작업 큐의 경쟁 문제를 해결하기 위해 제안된

AGSS(Adaptive Guided Self-Scheduling)는 백-오프 방법을 사용하여 반복을 갖고 오기 위해 경쟁하는 프로세서들의 개수를 줄이고, 각 반복들의 수행 시간이 다양하게 변할 수 있는 경우를 고려하여 연속된 반복들을 하나의 프로세서에 할당하는 것이 아니라, 서로 다른 프로세서에 각각 흩어지게 할당하므로 부하 불균형의 위험을 작게 하였다[8]. 이러한 예는, 루프 반복의 수행 시간이 선형적으로 증가 혹은 감소하거나 조건문이 포함된 경우이다. Factoring 기법 또한, 반복들의 수행 시간이 다양하게 변할 수 있는 경우에 부하 불균형의 위험을 줄일 수 있다[9]. 이 기법에서 루프 할당은 각 배치(batch)별로 이루어지며, GSS 할당 기법처럼 남은 반복의 전부를 고려하는 것이 아니라, 그 중 절반의 부분 집합만을 프로세서의 수로 나누어서 할당한다. 각 배치는 동일한 크기의 덩어리를 p번 아래와 같이 할당한다.

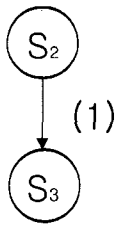
$$R_o = n,\ R_{i+1} = R_i - pF_i$$
$$F_i = \lceil R_i / x_i p \rceil,\quad x_i = 2 \qquad (2)$$

여기서, F_i 는 i 번째 배치의 스케줄링에 사용된 덩어리의 크기이다. Factoring 기법으로 스케줄링 하기 위해 x_i를 최적의 값인 2[9]로 하여 스케줄링한다. Factoring과 마찬가지로 Tapering은 계산량이 비 정규화한 루프를 위해 제안되었다[10]. 응용에 따라 예측 불가능한 조건문 등에 의해 수행 시간이 다양해 질 수 있는데, 이 알고리즘은 실행 프로파일 정보를 이용하여 효율적으로 스케줄링한다. TSS는 부하 균형을 유지하면서 스케줄링 오버헤드를 줄이기 위해 제안되었으며, 처음 몇몇 프로세서에게는 큰 덩어리인 $n/2p$ 만큼 할당하고 갈수록 $n/8p^2$ 만큼 크기가 다르도록 연속적으로 작게 할당하는 기법이다[11]. 지역 작업 큐 기반의 친화 스케줄링 기법은 바깥 루프가 순차 루프이고 안쪽은 병렬 루프를 가진 중첩된 루프 구조에 매우 효율적이다[12,13,14]. 루프에서의 어떤 반복은 특정한 프로세서에 대해 친화성을 가질 수 있다. 즉, 특정 프로세서의 지역 메모리나 캐쉬에 특정 반복의 수행에 필요한 자료가 존재할 경우 그 프로세서에 해당하는 반복을 할당한다. 이러한 스케줄링 기법을 이용하여 분산 환경에서 가장 비용이 비싼 통신 오버헤드를 감소하여 전체적인 성능을 향상시킬 수 있다[12,13,14]. 지금까지 살펴본 알고리즘들은 중앙 큐나 분산 큐를 이용하여 반복들이 독립적인 루프를 스케줄링 하는 기법들이다.

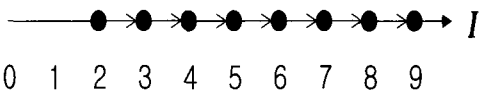
루프 캐리 종속성은 어떤 반복에서 사용된(read/write) 자료가 또한, 다른 반복들에 의해 사용되어질 때 발생하며, 자료 종속성 분석은 하나의 루프의 자료 흐름에 관한 정보를 제공한다. 만약, 반복들간에 자료 흐름이 발생하지 않다면 그 루프는 바로 병렬화가 가능하다. 루프 스케줄링 시, 하나의 반복은 프로세서에 할당되어지는 작업 단위이기 때문에 캐리 종속성은 루프의 병렬화에 큰 제약이 되며, 종속성 만족을 위한 동기화가 필요하다. 그림 1은 루프 캐리 종속성이 존재하는 예제 프로그램과 그에 해당하는 종속 그래프를 나타낸다.

```
S1: for I=2 to 9 do
S2:     A[I] = B[I] + C[I]
S3:     X[I] = A[I-1] + 2
S4: endfor
```

(a) 예제 프로그램
(a) An example program



(b) 자료 종속 그래프.
(b) data dependence graph



(c) 반복 공간 종속 그래프
(c) iteration space dependence graph

그림 1. 예제 프로그램과 종속 그래프
Fig. 1. An example program and its dependence graph.

그림 1(a) 예제 루프는 S_2와 S_3 문장간에 순방향 흐름 종속(forward flow dependence)관계를 가지며($S_2\delta^f S_3$), 종속 거리(dependence distance)는 1이다. 그림 1(b)는 (a)의 자료 종속 그래프(data dependence graph)를 나타내며, 괄호 안의 숫자는 종속 거리를 나타낸다. 그림

1(c)는 예제 프로그램의 반복 공간 종속 그래프(iteration space dependence graph)로, 각 반복들은 하나의 점으로 표현되며 반복의 어떤 문장이 루프의 다른 반복에 있는 문장에 종속적일 때, 종속 관계는 소스 반복에서 타겟 반복으로 간선에 의해 표현된다[1,2]. 또한, 루프 종속 그래프(loop dependence graph, 이하 LDG)로 반복들간의 종속 관계를 표현할 수 있다[1,2]. 반복들간에 종속 관계가 존재하는 루프에 대한 효율적인 셀프 스케줄링에 관한 연구가 필요함에 따라, 다음 Ⅲ장에서 본 논문에서 제안하는 새로운 할당 기법을 소개한다. 그리고, 중앙 큐 기반의 셀프 스케줄링 기법들을 루프 캐리 종속성을 가진 루프의 스케줄링에 적용하기 위한 그들의 변형에 대해 알아본다.

Ⅲ. 제안된 할당 기법

루프의 반복간에 종속성이 존재하지 않는 병렬 루프의 스케줄링은 루프의 프로세서들이 모든 반복이 끝날 때까지 중앙 작업 큐의 프론트로부터 순서대로 작업을 가져와서 수행한다. 하지만, 루프 캐리 종속성이 있는 루프를 병렬화 할 때는 고려해야 할 사항이 있다. 그것은 반복들간에 생기는 자료 흐름에 따른 동기화를 만족시키는 것으로, 프로세서가 수행을 위해 루프 반복들을 가져와도 그 반복의 종속 관계 만족 여부에 따라 수행 또는 대기 상태가 된다. 즉, 스케줄링 오버헤드를 줄이기 위해 큰 덩어리를 가져와도 덩어리의 처음 반복의 종속성이 만족 될 때까지 대기 상태가 된다.

본 논문에서 제안하는 동기화가 필요한 루프의 효율적 수행을 위한 새로운 할당 기법 CDSS(Carried-Dependence Self-Scheduling)는 알고리즘 1, 2와 같다. 여기서 d는 종속 거리이며, 종속 거리를 고려하여 루프

알고리즘 1. 제안한 CDSS 할당 기법의 입구부분
Algorithm 1. Entry block of the proposed CDSS assignment method

```
Entry Block :
    /* get_routine */
1.  lock(TaskQueue)
2.      Temp = TaskQueue
3.  unlock(TaskQueue)
4.  if (Temp[front] == 1) then
5.      i = get_from_queue(1)
                    /* 루프의 시작 부분 할당단계 */
6.      Execute Block(i)
7.  else if (Rest > d-1)
8.      start_chunk = get_from_queue(d)
                    /* 루프의 중간 부분 할당단계 */
9.  else
10.     start_chunk = get_from_queue(Rest)
                    /* 루프의 마지막 부분 할당단계 */
11. end if
12. update Rest     /* Rest 변수 갱신 */
```

알고리즘 2. 제안한 CDSS 기법의 검사, 수행, 출구부분
Algorithm 2. Test, execute and exit block of the proposed CDSS method

```
Test Block :
   /* exec_test_routine */
1.  for k=0, d-1
2.     i = start_chunk + k
3.     lock(CrossDep)
4.        Temp = CrossDep
5.     unlock(CrossDep)
  Exec_test:
6.     if (Temp[i] == 1) then
7.        Execute Block(i)
8.     else
9.        waiting for system_defined interval
10.       reload Temp from CrossDep
11.       Exec_test
12.    end if
13. end for
Execute Block(i) :
14.    execute the code of iteration i
Exit Block :
    /* fetch_set_routine */
15. j = detect(i)              /* j = i+d */
16. lock(CrossDep)
17.     fetch_set(j, 1)
18. unlock(CrossDep)
```

를 세 단계로 스케줄링한다. 첫 번째 한 번의 할당이 한 개의 반복을 가지는 초기 단계와 휴면 프로세서들이 종속 거리 d 개만큼의 덩어리를 가져오는 중간 단계 그리고, 남은 것이 d 보다 작을 때 전부를 가져오는 마지막 단계로 나눌 수 있다. 제안된 스케줄링 기법에서 프로세서는 크게 네 가지 기능을 수행한다. 먼저, 중앙 큐로부터 태스크를 가져오며, 가져온 태스크의 수행 가능 여부를 검사한 후, 조건이 만족되면 수행을 한다. 마지막으로, 현재 수행을 끝낸 반복에 종속적인 반복을 찾아 수행이 가능함을 알려 동기화 시킨다. CDSS 할당 기법의 수행을 위한 전제 조건으로는 루프 조절 변수의 다양한 초기, 증감, 최종 값을 루프 정규화를 통해 초기와 증감치가 1이라고 가정한다. 그리고, 루프 종속 관계를 나타내는 종속 거리는 컴파일 시간에 알 수 있는 상수라고 가정하며, 루프 각 반복들의 작업량이 동일(workload balancing)하여 수행 시간이 같다고 본다. 또한, 조건문에 의해 생기는 제어 종속성(control dependence)과 중첩된 루프는 고려하지 않는다. 알고리즘에 사용된 자료구조는 중앙 작업 큐(TaskQueue), CrossDep, Rest, Temp 변수이다. 임계 영역에 해당하는 공유 변수 CrossDep와 루프 반복들의 식별자를 가지는 TaskQueue는 잠금(locking)으로 직렬화(serialization)를 유지하며, 지역 변수로 사용된 Temp는 공유 메모리에 접근 해 있는 시간을 줄이기 위해 사용된다. Rest 변수는 현재 남은 반복의 수를 나타낸다.

알고리즘은 크게 네 부분으로 나누어진다. 첫 번째, 입구 부분(알고리즘 1의 줄 1-12)은 스케줄링을 위해 추가되어지며, 반복의 수행을 위해 작업 큐의 프론트로부터 루프 반복을 가져온다. 가져오는 반복 개수에 따라 스케줄링 횟수가 결정되며, 이것은 실행 시간 오버헤드와 밀접한 관계가 있다. 제안한 할당 기법에서 가져오는 덩어리의 크기는 루프의 시작, 중간, 끝 부분이 각각 틀리다. 알고리즘 1의 4-6번은 루프의 시작 부분 할당 단계로, 모든 루프의 첫 번째 반복은 종속성이 존재하지 않아 바로 수행 가능하기에 처음 하나의 반복만을 휴면 프로세서가 할당받아 수행한다. get_from_queue() 함수는 큐의 프론트로부터 인수 개수만큼의 반복을 가져오며 첫 번째 반복의 식별자를 리턴 한다. 알고리즘 1의 7-8번은 루프의 중간 부분 할당 단계로, 현재 남은 반복의 수(Rest)가 종속 거리(d) 보다 크거나 같을 때 큐로 부터 종속 거리인 d 개만큼의 반복들을 가져온 후, 덩어리의 첫 번째 반복 식별자를 start_chunk에 리턴 한다. 알고리즘 1의 9-10번은 루프의 마지막 부분 할당 단계로 남은 것이 d 보다 작을 때 남은 것 모두를 가져온다. 알고리즘 1의 12번은 가져온 덩어리를 현재 남은 반복의 수에 적용하기 위해 Rest 변수를 갱신한다.

두 번째로 검사 부분(알고리즘 2의 줄 1-13)에서는 가져온 덩어리의 첫 번째 반복부터 현재 수행 가능한지를 검사하는 루틴으로 종속 관계가 만족되었을 때 실행을 하게 된다. 각 반복들의 종속성 만족 여부를 표현하는 자료구조 CrossDep는 첨자가 1부터 시작되는 n (반복 수) 비트의 배열로, 각 원소에 저장된 값은 첨자에 해당하는 반복의 자료 종속성에 관계되는 정보이다. 즉, 이것은 루프의 i 번째 반복 L_i의 종속적 후행 반복인 L_j 의 수행 가능 여부를 결정한다. 만약, CrossDep의 j 번째 비트 값이 1이면, L_j의 선행 노드인 L_i의 수행이 끝나 L_j의 자료 종속성이 만족되어 현재 L_j가 수행 가능함을 의미한다. 초기 값으로, 루프 종속 그래프(LDG)에서 들어오는 종속 아크가 없는 반복들은 모두 1로 셋(set) 되어진다. 즉, 이러한 반복들은 독립적이기에 바로 수행 할 수 있으며 나머지는 모두 0으로 채워진다. 임계 영역에 접근해 있는 시간을 줄이기 위해 CrossDep 내용을 임시 변수에 저장하며, i 번째 비트가 1이면 i 번째 반복의 종속성이 만족되었기에 반복을 수행한다(알고리즘 2의 줄 6-7). 만약, 종속성이 만족 되지 않은 경우는 대기 상태가 되어 시스템이 지정한 간격 동안 대기하며 갱신된 CrossDep에 다시 접근하여 검사 과정을 반복한다(알고리즘 2의 줄 8-12). 갱신된 CrossDep를 재 적재하여 실행 가능 여부를 체크하는 주기는 실행 시간 오버헤드와 관계되어 루프 수행 시간에 영향을 주게된다.

다음으로, 수행 부분(알고리즘 2의 줄 14)에서는 루프 몸체의 코드를 수행한다. 마지막으로, 출구 부분(알고리즘 2의 줄 15-18)에서는 현재 수행이 끝난 반복을 CrossDep에 반영하기 위해 반복 L_i 의 수행이 끝나면, 그것에 종속적인 L_j를 $i+d$ 연산으로 찾아(detect(i)) 해당하는 CrossDep 비트를 1로 만든다(fetch_set(j, 1)). 이것은 L_j가 현재 수행 가능함을 동기화 시킨다.

제 II장에서 소개한 중앙 큐 기반의 셀프 스케줄링 기법들을 이용하여 동기화가 필요한 루프 캐리 종속성을 가진 루프의 병렬 수행을 위해서는 기존 알고리즘

들의 변형이 필요하다. 먼저, 각 프로세서가 수행을 위해 가져온 반복들의 종속성이 만족되어 현재 수행 가능한지 검사하는 부분이 추가 되어야하며, 이러한 연산 결과에 따라 가져온 반복이 수행 또는 대기 상태가 된다. 그리고 만약, 수행 가능하다면 수행 후, 그것에 종속적인 후행 반복을 찾아 종속성이 만족되어 현재 수행 가능함을 알리는 루틴 또한 필요하다. 지금까지 설명한 알고리즘의 변형이 필요한 부분은 제안한 기법의 일부인 알고리즘 2를 각 알고리즘에 동일하게 추가 적용시키면 된다. 여기서, 알고리즘 2의 1번 줄에 있는 종속 거리를 나타내는 d를 각 스케쥴링 정책에 따라 단계별로 가져오는 덩어리의 크기로 바꾸어 주어야한다. 그리고, 종속성 만족 여부를 표현하기 위해서 자료구조 CrossDep가 필요하다.

Ⅳ. 알고리즘의 구현 및 성능 평가

1. 알고리즘의 구현

본 논문에서 제안된 CDSS 기법과 네 개의 중앙 큐 기반의 셀프 스케쥴링 알고리즘을 Ⅲ장에서 설명한 방법대로 변형 후, 단일 처리기 시스템을 포함한 여러 플랫폼에서 수행하기 위해 JDK 1.2.2와 통합 개발 패키지인 Kawa3.22를 이용하여 프로세서의 셀프 스케쥴링 기능을 스레드 레벨로 구현하였다. 자바는 두 종류의 스레드를 제공하며 실험에 사용된 것은 그 중, 그린 스레드(green thread)로 스레드 대 프로세서의 매핑은 M:1로 이루어진다[15]. 임계 영역의 동기화 구현은 잠금을 위한 synchronized 키워드와 wait, notifyAll 등의 동기화 메소드를 사용하였다. 다음은 제안된 기법의 구현과 기존의 네 개 기법들을 루프 캐리 종속성이 있는 루프의 스케쥴링을 위해 설계된 클래스들이다.

○ Processor ; 프로세서가 수행할 기능들을 스레드로 구현한 것으로, Thread 클래스로부터 상속받는다. 하나의 일을 하는 시간은 특정 시간을 미리 정해서(구현에서는 30msec), 이 시간을 작업하는 시간으로 간주하였다. 만약, 종속성이 해결되지 않았다면, 스레드를 멈추고 다음 순서가 될 때까지 기다렸다가 다시 종속성을 검사하게 된다. 이 시간은 고정적으로 결정되어 있지 않다. 실험 시스템(Redhat Linux 6.1)에서 기본적으로 타임 슬라이싱을 하고 있으므로, 어떠한 작업이 타임 슬라이싱보다 작다면 여러 개의 일을 할 수도 있다. 스케쥴링 단위인 청크를 중앙 큐에서 즉, Scheduler 클래스에서 가져와 이 클래스에 의해 반복을 수행한다. 수행 시간은 각 반복의 의존성 검사에 따른 시간과 청크를 중앙 큐에서 가져오는데 걸리는 오버헤드를 포함한다.

● Job ; 루프 반복들의 현재 만족된 종속성 정보를 표현하기 위해 CrossDep 배열을 생성 후, 각 반복의 수행이 끝남에 따라 이 정보가 갱신된다. 여러 스레드가 CrossDep에 접근하여 읽기, 쓰기를 하기에 동기화가 필요하다. 우리의 작업은 종속성이 있기 때문에 현재 종속성이 어디까지 해결이 되었는지를 알 수 있어야 한다. 이를 위해 생성자는 작업이 몇 개인지를 알고 그만큼 리스트를 생성한 후, 작업이 끝난 반복의 종속적인 반복의 원소에 1로 표시하고, 종속성이 아직 해결되지 않은 것은 0으로 하여 구분을 하였다.

● Scheduler ; 응용 프로그램과 시스템 파라메터 값에 따른 스케쥴링을 위한 초기화 작업을 한다. 스케쥴링 기법 SS, CSS, GSS, Factoring, CDSS 에 따라 청크 내에 들어갈 작업의 크기와 순서, 개수를 계산하여 청크 리스트를 만든다. 따라서, Processor 클래스는 작업 큐에 접근하는 순서대로 아직 가져가지 않은 청크만 가져가게 하였다. 각각의 청크는 그 청크내의 시작 작업 번호와 마지막 작업 번호를 가지고 있어서, Processor 클래스는 이 번호만 보고 작업을 진행할 수 있다.

● Tester ; 파라메터 값에 따라 Scheduler를 생성하여 작업 스케쥴링을 하고 각각의 스레드를 실행시킨다.

2. 알고리즘의 성능 평가

제안된 CDSS 기법의 성능을 평가하기 위해 변형된 SS(Modified Self-Scheduling,이하, MSS), CSS(이하, MCSS), Factoring(이하, MFact)과 GSS(이하, MGSS)의 네 개 셀프 스케쥴링 알고리즘을 비교 대상으로 선택하였다. 알고리즘의 성능평가는 스케쥴링 오버헤드와 동기화에 따른 지연 시간을 포함한 루프 전체의 수행 시간을 측정한다. 시뮬레이션에 사용된 응용 프로그램의 파라메터인 작업노드의 수(n)는 60, 120, 180개로 변화시켰다. 본 논문에서의 실험 대상인 루프는 반복들의 수행 시간이 루프 조절 변수 값의 변화에 따라 증가 혹은, 감소하지 않는 동일한 연산 시간을 가진다. 따라서, LDG[1,2]의 각 노드가 가지는 실행 비용(e)을 동일하게 각각 20, 70, 120, 170, 220msec로 변화하였다. 그리고, LDG에서 간선으로 표현되는 자료 종속성의 종속

거리(d)는 2, 3, 4인 응용이다. 시스템 파라메터 값으로 스레드 수(t)는 2, 3, 4, 10, 20, 30개로 생성하였으며, 실험 대상이 파인 그레인(fine grain)의 태스크이기에 실행은 메모리 128Mb를 가진 Pentium-III 450MHz (Redhat Linux 6.1)의 단일처리기 시스템에서 하였다. 스레드 수, 종속 거리, 태스크 수, 태스크 크기의 변화에 따라 제안된 스케줄링 기법과 변형된 네 개 알고리즘들의 루프 수행 시간의 측정 결과는 표 1과 같다.

표 1. 스케줄링 기법별 수행시간 (단위:msec)

Table 1. The execution time for scheduling algorithms(unit: miliseconds)

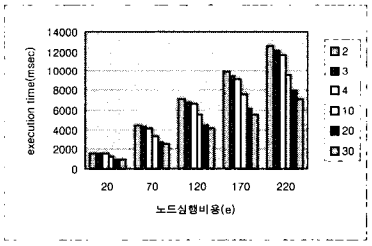
t	d	n	e	Schedule Style				
				MGSS	MFact.	MSS	MCSS	CDSS
2	2	60	20	1627	1537.2	888.8	1726.2	888.8
			120	7040.4	6656.8	3844.4	7540.6	3839.6
	3	120	170	19726.6	18284.4	10765.4	21158.4	10754.8
			220	25618	23390.6	13756.8	27052.4	13749
	4	180	20	4819.2	4508.2	2660.2	5202.6	2636.4
			120	20982.8	19624.6	11514.6	22606.6	11476.8
3	2	120	70	8792.2	8416.4	4775	9349.2	4768.4
			120	14199	13562.8	7742.4	15056.8	7714.8
	3	60	20	1386.6	1170.4	586.4	1646.4	588.8
			70	3727.8	3169	1588.6	4440.6	1584.8
	4	180	170	27668.8	25302.4	10759.8	31212.6	10764.8
			220	35322.8	32366.2	13766.2	39904.6	13754.8
4	2	60	70	4054.2	3734.4	2384	4522.8	2380.4
			170	9153.2	8434.6	5392	10225.6	5388.6
	3	120	20	2883.2	2665	1180.2	3371.4	1176.6
			120	12539	11543.4	5157.8	14628.2	5154
	4	180	20	4356.4	3976.2	1323.4	5052.8	1338.6
			220	33948.4	30974	10306.8	39220.8	10329.2
10	2	180	170	27252.2	25282.4	16174.6	30688.6	16172.2
			220	34845.8	32320.8	20680.2	39229.8	20669
	3	120	20	2281.8	1876.2	1179.8	2959.6	1170
			120	10044.6	8216.4	5159.4	13085.2	5144.4
	4	180	120	14763	12042.2	5834.8	19717	5800
			170	20463.8	16702.8	8080.4	27440	8050.6
20	2	120	20	2416.6	2110	1788.8	2972.8	1762.6
			70	6522.8	5653.6	4793.4	8025.2	4756.4
	3	60	20	698.6	765.2	592.8	652	592.6
			170	4301.2	4661.6	3588.2	3940.6	3576.8
	4	180	120	10900.2	8184.2	5845.4	15901.2	5835.8
			220	19287.2	14469.8	10341.2	28213.6	10340
30	2	180	20	3666.2	3173.8	2695.2	4495.8	2666.6
			70	9814.2	8471.6	7195.2	12034	7159.6
	3	120	170	9403.8	9000.4	7194.6	11117.6	7196.8
			220	11986	11487.8	9195.6	14223.8	9187.6
	4	60	120	2080.8	1951	1951.8	2076.4	1949.6
			170	2874	2699.8	2703	2880.8	2693.4

표는 실험 결과의 일부를 나타낸 것으로 시간 단위는 밀리초(msec)이며, 각 값은 동일한 파라메터 값에 대하여 10번 수행한 결과의 평균 시간이다. 수행 시간은 스케줄링을 위한 초기화 단계와 스레드의 생성 시간은 측정에서 제외되었으며, 스레드를 런 상태로 만든 시점부터 스레드들이 LDG의 마지막 작업 노드를 끝내는 시점까지의 경과 시각으로 측정하였다. CDSS기법을 제외한 나머지 알고리즘들은 파라메터 값의 변화에 따라 서로간의 성능 우수성이 바뀌는 것을 알 수 있다. 하지만, 본 논문에서 제안한 CDSS 할당 기법은 다양한 실험 환경에서 대부분의 경우 최소의 작업 시간을 유지하여 수행 시간 측면에서 효율적이다. 실험 파라메터 값에 대해 제안된 기법은 MSS, MFact, MGSS, MCSS 기법에 대해 각각 0.02, 40.5, 46.1, 53.6%의 성능 향상을 보였다. 다음은 응용 프로그램과 시스템 파라메터 값의 변화에 따른 수행 시간과의 관계이다.

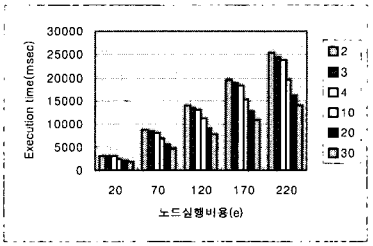
2.1. 시스템 파라메터

병렬 처리를 위해 스레드 개수를 동일한 응용에 대해 최대 30개까지 생성시켰다. 그림 2는 스케줄링 기법별 스레드 수에 따른 수행시간을 나타낸 것이다.

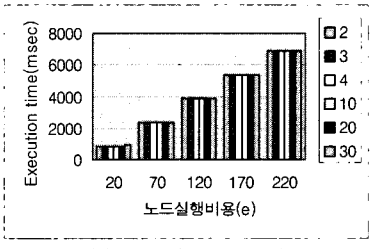
그림 2(a)는 d=2, n=60인 응용에 대해 루프 반복의 실행 비용(e)을 20에서 220msec로 증가시킨 경우의 스레드 수에 따른 MGSS 기법의 수행 결과이다. MGSS에서는 d=30, n=60인 경우에만 30개 스레드를 투입시, 성



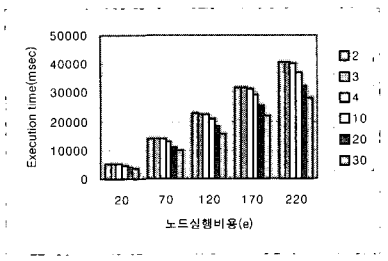
(a) MGSS(d=2, n=60)



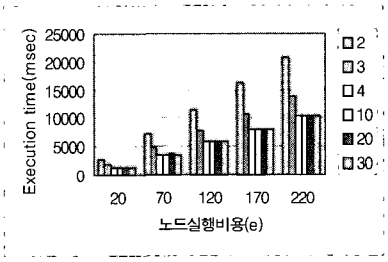
(b) MFact(d=2, n=120)



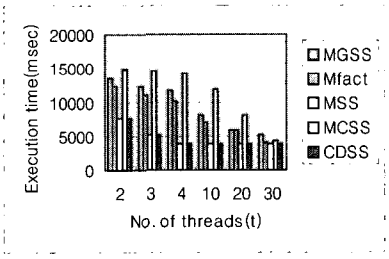
(c) MSS(d=2, n=60)



(d) MCSS(d=3, n=180)



(e) CDSS(d=4, n=180)



(f)정책별 수행시간(d=4, n=120, e=120)

그림 2. 스레드 수와 수행시간
Fig. 2. The number of threads and execution time

능 저하를 보였으며 나머지 모든 경우는 많은 수의 스레드를 생성하여 수행 시간을 감소시킨다. 즉, 응용 프로그램의 파라메터 값에 관계없이 대부분의 경우 많은 스레드를 생성함으로 성능 향상을 추구할 수 있다.

그림 2(b)는 MFact의 결과로써, 대부분의 경우 스레드 수를 증가하여 성능 향상을 보였으며, d=3, n=60인 응용인 경우 스레드를 20개 생성할 때, 성능 저하를 보이지만 30개를 사용한 경우는 수행 시간을 줄일 수 있었다. 예로서, d=3, n=60, e=220인 경우, 2, 3, 4, 10, 20, 30개의 스레드를 실행시켜 각각 10553, 9178, 8716, 5507, 5963, 4595msec로 수행 시간을 줄인다. 그리고, d=3, n=120과 d=4, n=180인 경우는 20개 스레드를 투입하여 최대 성능을 얻을 수 있었다.

MSS(그림 2(c))에서는 성능 향상을 위해 생성된 스레드 수는 종속 거리와 관계가 있음을 알 수 있다. 종속 거리가 2인 응용인 경우, 대부분 2개 스레드로 최대 성능 얻으나, n=60인 경우는 20개로 최대 성능을 얻을 수 있었다. 종속 거리가 3일 때는 3개 스레드를 사용하여 대부분 효율성을 높일 수 있었으나, n=60, e=120인 경우는 4개 스레드가 적당했다. 그리고, d=4인 경우, 4개 스레드를 사용함으로 대체적으로 수행 시간을 최대로 줄일 수 있었으나, n=120, e=20과 n=180, e=20과 e=170인 경우는 10개가 적절했다. 또한, MSS에서는 30개 이상의 스레드를 생성하여도 성능 향상에 도움을 주지 못했다. 반면에, 같은 파라메터 값인 그림 2(a)와 비교할 때, MGSS는 30개 스레드를 사용할 때 가장 효율적이다.

MCSS 기법은 그림 2(d)에서 보듯이, 스레드 수 증가에 따라 수행 시간이 현저히 감소함을 알 수 있으며, d=3, n=60인 응용에만 20개가 효율적이며 나머지 모든 경우는 최대의 스레드 개수인 30개를 사용함으로 수행 시간을 줄일 수 있었다. 그림 2(e)에서 보듯이, 제안된 CDSS에서는 성능 향상을 위해 생성된 스레드 수는 종속 거리와 밀접한 관계가 있다. 대부분의 경우, 종속 거리에 해당하는 스레드 수를 사용하는 것이 가장 효율적이다. 그림 2(f)는 d=4, n=120, e=120일 때, 각 스케쥴링 기법별 스레드 수에 따른 수행 시간을 나타낸 그래프이다. 여기서, 제안된 기법과 MSS가 좋은 성능을 보이며, 두 할당 기법 모두 종속 거리가 4인 응용에 대해 종속 거리에 해당하는 4개의 스레드를 사용할 때 효율적임을 알 수 있다.

실험 결과, 대체적으로 스레드 수를 증가할수록 루프 수행 시간을 줄여 효율적임을 알 수 있었다. 하지만, 스레드 수를 증가시킴에도 불구하고 성능이 떨어지는 경우도 있었다(그림 2(c)). 스레드 수에 따른 성능 향상은 각 스케쥴링 기법과 파라메터 값에 따라 종속적임을 실험을 통해 알 수 있었다. 그리고, CDSS와 MSS에서 성능 향상을 위해 사용되는 최적의 스레드 수는 종속 거리와 관계가 있어 d 개의 스레드를 사용할 때 효율적이다.

2.2. 응용 프로그램의 파라메터

수행 시간에 영향을 미치는 응용 프로그램의 인자들 중에 작업량과 태스크 크기, 그리고 종속 거리의 변화에 따른 수행 시간을 알아보고자 한다. 먼저, 작업량의 변화로, 태스크 수가 증가됨에 따라 동일한 개수의 스레드를 사용한 경우 수행 시간이 길어졌다. 그림 3은

d=4, e=70, t=30인 파라메터 값에 대해 반복 수(n)의 변화에 따른 각 할당 기법별 루프의 수행 시간이다. 제안된 CDSS 기법에서 반복 수를 60, 120, 180개로 증가함에 따라 수행 시간은 각각 1201, 2373, 3577msec로 길어졌다. 즉, 노드 수가 두 배로 증가함에 따라, 비례적으로 약 두 배 정도로 수행 시간이 걸렸다.

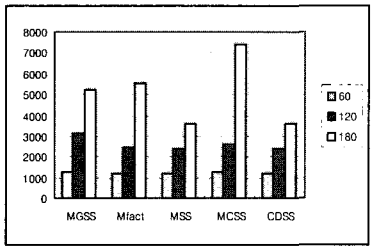


그림 3. 반복 수와 수행시간(d=4, e=70, t=30)
Fig. 3. The number of iterations and execution time

다음으로, 작업의 크기, 즉 노드의 실행 비용(e)을 변화시켜 보았다. 그림 4는 d=3,n=120,t=10인 경우에 대한 태스크 크기 변화에 따른 수행시간을 나타낸다. 예로서, 노드 수가 120개인 LDG에 대해 노드 당 연산 시간을 20, 70, 120, 170, 220msec로 변화시켜 제안된 알고리즘을 이용해 실험한 결과, 수행 시간이 각각 1170, 3173, 5144, 7173, 9160msec로 길어졌다. 결과적으로, 응용 프로그램의 파라메터인 n, e 값이 커짐에 따라 이에 비례하여 수행 시간이 증가한다.

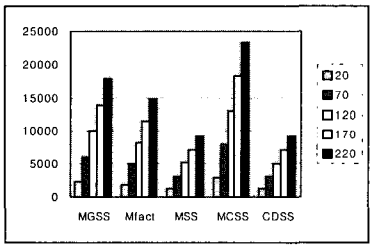


그림 4. 태스크 크기와 수행시간(d=3, n=120, t=10)
Fig. 4. Task size and execution time

마지막으로, 그림 5는 n=120,e=20인 응용에 대해 20개의 스레드를 실행시켰을 때, 종속 거리(d)의 변화에 따른 수행시간을 나타낸다. 이 실험에서는 종속 거리가 큰 응용일수록 수행 시간이 적게 걸렸다. 예로서, CDSS 기법을 적용하여 d 가 2, 3, 4인 경우에 대해 수행 시간은 각각 1763, 1166, 891msec로 짧아졌다. 이것은 종속 거리가 큰 응용일수록 만족해야 할 종속성의 개수가 적어 동기화 만족을 위한 지연시간이 줄어들고, 같은 시각에 동시 수행 가능한 스레드의 수가 많아 스레드의 병렬성이 많아지기 때문이다. 지금까지의 다양한 파라메터 값의 변화에 따른 실험 결과, 할당 기법별로는 CDSS, MSS, MFact, MGSS, MCSS 순으로 수행 시간 측면에서 성능이 우수하다.

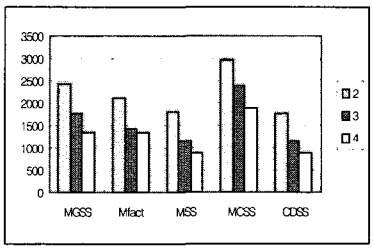


그림 5. 종속거리와 수행시간(n=120, e=20, t=20)
Fig. 5. Dependence distance and execution time

V. 결 론

본 논문에서는 루프 반복들간에 의존성이 있는 루프를 공유 메모리 다중처리기 환경에서 효율적으로 수행하기 위한 중앙 큐 기반의 새로운 셀프 스케줄링 기법을 제안하였다. 또한, 지금까지 소개된 중앙 큐 기반의 대표적인 병렬 루프의 셀프 스케줄링 기법들을 동기화가 필요한 루프 캐리 종속성을 가진 루프의 할당에 적용하기 위한 그들의 변형에 대해 연구하였다. 단일처리기의 멀티스레드 환경을 포함한 고성능의 병렬 시스템에서 수행하기 위해 제안된 기법과 변형된 알고리즘들을 자바 스레드를 이용하여 구현하였다. 응용 프로그램과 시스템 파라메터 값을 다양하게 하여 변형된 기법들과 스케줄링 오버헤드와 동기화에 따른 지연시간을 포함한 전체 루프 수행 시간을 비교 분석한 결과, 제안된 CDSS 기법이 동기화에 따른 지연 시간을 줄여 다른 변형된 기법들에 비해 수행 시간 측면에서 효율적이다. 할당 기법별로는 MSS, MFact, MGSS, MCSS 순으로 성능이 우수하였다. 대체적으로, 많은 수의 스레드를 생성하여 수행 시간을 줄일 수 있었지만, 성능 향상을 위한 최적의 스레드 개수는 사용된 스케줄링 기법과 특정 응용에 종속적임을 알 수 있었다. 그리고, 다른 기법들에 비해 제안된 기법에서는 성능 향상을 위해 사용된 스레드 수는 종속 거리와 밀접한 관계가 있어 대부분의 경우, 종속 거리에 해당하는 적은 수의 스

레드를 사용함으로 최대 성능을 얻는다. 향후 연구는 코오스 그레인(coarse grain)의 태스크를 가진 계산량이 많은 루프에 대해 고성능의 다중처리기 시스템에서 제안된 기법과 여러 스케쥴링 기법들의 성능을 비교 분석하고자 한다.

참 고 문 헌

[1] M. J. Quinn, Parallel Computing -Theory and Practice, McGraw-Hill, 1994.

[2] M. Wolfe, High Performance Compilers for Parallel Computing, Addison-Wesley, 1996.

[3] H. E. Bal and M. Haines, "Approaches for Integrating Task and Data Parallelism," IEEE Concurrency, vol.6, no.3, pp.74~84, 1998.

[4] C. P. Kruskal and A. Weiss, "Allocating independent subtasks on parallel processors", IEEE Trans. Software Eng. vol.11, pp.1001~1016, 1985.

[5] P. Tang and P. C. Yew, "Processor Self-Scheduling for multiple nested parallel loops," Proc. 1986 Int. Conf. Parallel Processing, pp.528~535, 1986.

[6] Z. Fang, P. Tang, P. C. Yew, and C. Q. Zhu, "Dynamic Processor Self-Scheduling for General Parallel Nested Loops," IEEE Trans. on Computers, vol.39, no.7, pp.919~929, 1990.

[7] C. D. Polychronopoulos and D. Kuck, "Guided Self-Scheduling : A Practical Scheme for Parallel Supercomputers," IEEE Trans. on Computers, vol.36, no.12, pp.1425~1439, 1987.

[8] D. L. Eager and J. Zahorjan, "Adaptive guided self-scheduling," Tech. Rep. 92-01-01. Dept. of Comput. Sci. and Eng,, univ. of Wash., 1992.

[9] S. E. Hummel, E. Schonberg, and L. E. Flynn, "Factoring : A Method for Scheduling Parallel Loops," Comm.ACM, vol.35, no.8, pp.90~101, 1992.

[10] S. Lucco, "A Dynamic Scheduling Method for irregular parallel Programs," Proc. ACM SIGPLAN '92 Conf. Programming Language Design and Implementation, pp.200~211, 1992.

[11] T. H. Tzen and L. M. Ni, "Trapezoid self-scheduling : A practical scheduling scheme for parallel computer," IEEE Trans. on Parallel and Distributed Syst., vol.4, pp.87~98, 1993.

[12] E. P. Markatos and T. J. LeBlanc, "Using Processor Affinity in Loop Scheduling on Shared-Memory Multiprocessors," IEEE Trans. on Parallel and Distributed Syst, vol.5, no.4, pp.379~400, 1994.

[13] S. Subramaniam and D. L. Eager, "Affinity Scheduling of Unbalanced Workloads," Proc. Supercomputing '94, pp.214~226, 1994.

[14] Y. Yan, C. Jin and X. Zhang, "Adaptively Scheduling Parallel Loops in Distributed Shared -Memory Systems," IEEE Trans. on Parallel and Distributed Syst., vol.8, no.1, pp.70~81, 1997.

[15] M. Campione, The Java Tutorial, Addison-Wesley, 1999.

저 자 소 개

金 炫 澈(正會員)

1995년 : 경일대학교 컴퓨터공학과 졸업(공학사). 1997년 : 경북대학교 컴퓨터공학과 졸업(공학석사). 1999년 : 경북대학교 컴퓨터공학과 박사수료. 2000년~현재 : 포항1대학 정보통신과 전임강사. <관심분야> 어레이 프로세서 설계, 병렬 및 분산처리, 암호화 등

金 孝 喆(正會員)

1987년 : 경북대학교 전자공학과(전산) 졸업(공학사). 1989년 : 경북대학교 전자공학과(전산) 졸업(공학석사). 1989년~1996년 : 국방과학연구소 근무. 1999년 : 경북대학교 컴퓨터공학과 박사수료. 1996년~현재 : 계명문화대학 조교수. <관심분야> 멀티미디어, 패턴인식, 암호화 등

柳 基 永(正會員)
1976년 : 경북대학교 수학교육학과 졸업(이학사). 1978년 : 한국과학기술원 전산학과 졸업(공학석사). 1992년 : 미국 Rensselaer Polytechnic Institute 졸업(이학박사). 1978년~현재 : 경북대학교 컴퓨터공학과에 재직. <관심분야> 병렬처리, DSP array processor 설계, 암호화 등