루프 캐리 종속성을 가진 루프의 할당 기법
(An Assignment Method for Loop with Loop-Carried Dependence)

김 현 철 † 유 기 영 ††
(Hyun-Chul Kim) (Kee-Young Yoo)

요 약 본 논문에서는 루프 반복들 간에 종속 관계가 존재하는 루프의 효율적 수행을 위한 새로운 루프 할당 기법을 제안한다. 그리고, 중앙 큐를 사용하여 공유 메모리 다중처리기에 루프 반복을 할당하는 기존 셀프 스케줄링 기법들을 루프 캐리 종속성(loop-carried dependence)을 가진 루프의 할당에 적용하기 위해 제안한 기법을 이용한 그들의 변형에 대해 알아본다. 종속 거리를 고려하여 루프를 세 단계별로 할당하는 제안된 CDSS (Carried-Dependence Self-Scheduling) 기법 또한, 중앙 작업 큐를 기반으로 한 것이며 별도의 스케줄러가 필요 없는 셀프 스케줄링 알고리즘이다. 종속 거리, 프로세서 수, 반복 수, 스케줄링 연산 시간 등을 다양하게 하여 변형된 할당 기법들과 비교 분석한 결과, 제안한 기법은 양호한 부하 균형을 유지하였으며 변형된 다른 기법들에 비해 루프 수행 시간을 줄여 효율적임을 알 수 있었다. 다양한 실험 환경에서 평균적으로 제안한 CDSS, 변형된 SS, Factoring, GSS, CSS 기법 순으로 루프 수행 시간 측면에서 좋은 성능을 보였다.

Abstract In this paper, we propose a new loop assignment method for the efficient execution of a loop with dependence between iterations. Moreover, the modification of existing self-scheduling schemes that assign loop iterations to shared-memory multiprocessors using a central job queue is suggested for scheduling a loop with carried-dependence using our scheme. The proposed CDSS (Carried-Dependence Self-Scheduling) method that assigns loops in three-level considering the dependence distance of loops is self-scheduling algorithm based on the central queue and additional scheduler is not necessary. Compared to other modified assignment algorithms with various changes such as dependence distance, the number of processors, the number of iterations, scheduling computation time, CDSS is found to be more efficient than other methods in load balance and execution time. In various experiment environments, CDSS, modified SS, Factoring, GSS and CSS are executed efficiently in order of execution time.

1. 서 론

병렬 처리에서 프로그램의 병렬성은 성김도(granularity)의 크기에 따라 크게 두 가지로 나눌 수 있다. 그 중 하나는, 작은 성김도(fine grain)의 태스크 병렬성 혹은 함수 병렬성이며, 다른 하나는 성김도가 큰(coarse grain) 경우인 자료 병렬성 또는 루프 병렬성이다. 루프는 과학 분야 등 많은 응용에서 널리 사용되며 가장 많은 병렬성을 제공하는 프로그램 구조이다. 자료 병렬성 추출을 위해 지금까지 연구된 대부분의 알고리즘들은 공유 메모리 다중 처리기 구조에서 독립적인 반복을 가진 루프를 효율적으로 할당하는 기법들이다. 이러한 할당 기법들은 스케줄링 오버헤드를 최소화하여 루프 전체의 수행 시간을 줄이며 또한, 좋은 부하 균형을 유지하므로 프로세서 각각의 성능을 높이는데 목표를 두고 있다. 분산 메모리나 분산 공유 메모리(DSM) 시스템에서의 스케줄링 시에는 원격 자료를 접근함으로 생기는 통신 오버헤드를 최소화 해야한다[1,2].

루프의 병렬 수행을 위해 반복들을 실행 시간에 다중 프로세서에 할당하는 루프의 동적 스케줄링 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 또는

† 비 회 원 : 포항1대학 정보통신과 교수
hckim@pohang.ac.kr
†† 종신회원 : 경북대학교 컴퓨터공학과 교수
yook@knu.ac.kr
논문접수 : 2000년 4월 17일
심사완료 : 2001년 7월 10일

셀프 스케쥴러 기법으로 구분된다. 셀프 스케쥴링은 병렬 컴파일러에 의해 생성되는 동적 셀프 스케쥴링 코드인 드라이브 코드(drive code)가 삽입됨으로써 이루어진다[1]. 프로세서가 수행 코드의 앞, 뒤 부분에 스케쥴링을 위해 추가되어진 코드 부분도 수행함으로써, 스케쥴러가 분산된 개념을 가지며 별도의 고정된 전역 스케쥴러 프로세서가 필요하지 않다. 즉, 프로세서가 코드 실행과 스케쥴링 기능을 병행하는 것이다. 중앙 큐를 사용하여 독립적인 반복을 가진 병렬 루프를 공유 메모리 다중처리기에 할당하는 셀프 스케쥴링 기법들은 SS (Self-Scheduling)[3], CSS(Chunk Self-Scheduling)[4], GSS(Guided Self-Scheduling)[5], Factoring[6] 등이다.

본 논문에서는 루프 반복들간에 종속 관계가 존재하는 루프의 효율적인 셀프 스케쥴링에 관한 연구가 필요함에 따라 종속성이 존재하는 루프를 효율적으로 수행하기 위해 새로운 할당 기법을 제안한다. 그리고, 위의 네 가지 중앙 큐 기반의 셀프 스케쥴링 기법들을 루프 캐리 종속성(loop-carried dependence)을 가진 루프의 할당에 적용하기 위해 제안된 알고리즘을 이용한 변형에 대해 알아본다. 제안된 CDSS(Carried-Dependence Self-Scheduling) 기법 또한, 중앙 작업 큐를 기반으로 한 것이며 별도의 스케쥴러가 필요 없는 셀프 스케쥴링 기법이다. CDSS는 종속 거리(dependence distance)에 따른 종속 관계를 만족시키기 위한 동기화 시점을 고려하여 루프를 새 단계별로 할당한다. 성능 분석은 종속 거리(d), 루프 반복의 수(n), 프로세서 수(p)와 스케쥴링 연산 시간을 다양하게 하여 변형된 할당 정책들과 제안한 기법의 루프 수행 시간 스텝, 부하 균형도, 동기화에 따른 지연 시간과 프로세서 병렬성을 시뮬레이션 하였다.

본 논문의 구성은 다음과 같다. 2장에서는 루프 할당 기법의 관련 연구를 살펴보고, 3장에서는 본 논문에서 제안한 루프 캐리 종속성을 가진 루프의 할당 기법인 CDSS와 기존 알고리즘의 변형에 대해 설명하며, 4장에서는 중앙 큐를 기반으로 하는 변형된 셀프 스케쥴링 알고리즘과 제안한 할당 기법을 비교 분석하며, 마지막으로 결론에 대해 기술한다.

2. 관련 연구

공유 메모리 다중 처리기 환경에서의 루프 스케쥴링 알고리즘의 우수성을 평가하는 요소는 스케쥴링 오버헤드와 부하 불균형이다. 최적의 스케쥴링 알고리즘은 위의 요소들을 모두 최소화하는 것이지만, 서로 상충 관계(trade-offs)가 있기에 두 가지 모두를 만족시키기는 불가능하다. 예로서, 프로세서의 성능이 각각 다르거나 루프 반복들의 계산량이 선형적으로 증가 혹은 감소하는 경우의 응용에서는 한 번 할당시 작은 양의 반복을 가질수록 좋은 부하 균형을 얻지만, 많은 양의 반복을 가져오는 경우에 비해 스케쥴링 횟수가 많아짐에 따라 스케쥴링 오버헤드가 증가된다.

루프의 반복을 프로세서에 할당하는 시점에 따라 루프 할당 기법은 크게 두 가지로 나눌 수 있다. 하나는 루프 반복을 컴파일 시간에 분배하는 정적 스케쥴링(static scheduling)으로 루프가 수행할 작업량을 모든 프로세서에 균등히 할당함으로써 최대 성능을 얻는다. 이때, 각 반복이 수행할 계산량과 사용할 프로세서 수를 미리 알아야 하지만, 루프는 조건 분기와 다양한 바운드의 내부 루프를 포함하기 때문에 각 반복의 수행 시간은 매우 다양하게 변화될 수 있어 대부분의 응용에 정적 스케쥴링이 사용되지 못하고 있다. 정적 스케쥴링은 루프의 시작점에서 오직 한번만 작업을 할당하기에 스케쥴링 오버헤드를 줄여 좋은 효율성을 가진다. 반면에, 실행 시간에 루프 반복을 할당하는 동적 스케쥴링(dynamic scheduling)은 많은 스케쥴링 연산의 오버헤드 때문에 효율성은 떨어지지만 좋은 부하 균형을 유지할 수 있다. 동적 스케쥴링 기법은 사용된 작업 큐와 전용 스케쥴러의 존재 여부에 따라 여러 가지로 세분화된다. 중앙 큐 기반의 셀프 스케쥴링 기법으로는 SS, GSS, Factoring, CSS, TSS (Trapezoid Self-Scheduling)[7]등이 있으며, 지역 큐를 사용하는 셀프 스케쥴링 기법으로는 친화 스케쥴링 (Affinity Scheduling)[8], SSS(Safe Self-Scheduling)[9]등이 있다. 지금까지 연구된 셀프 스케쥴링 알고리즘에 대해서 살펴보고자 한다.

SS 알고리즘은 병렬 루프의 모든 반복이 수행되어질 때까지 휴면 프로세서들이 하나의 반복만을 가져와 수행한다. 거의 완벽한 작업 균등을 유지하는 반면에, 각 프로세서가 임계 구역인 중앙 작업 큐에 반복 개수만큼 접근하기에 많은 스케쥴링 오버헤드를 발생시킨다. CSS는 한 번에 k개의 반복을 가져옴으로 스케쥴링 오버헤드를 줄이지만 부하 균형은 SS보다 좋지 못하다. GSS는 루프의 시작 부분에서는 큰 덩어리(chunk)를 할당하여 스케쥴링 오버헤드를 감소시키며 루프의 끝 부분에서는 작게 할당하여 부하 균형을 좋게 한다. 단점으로는, 루프 끝 부분을 수행하면서 작업 큐의 경쟁을 유발한다. 경쟁 문제를 해결하기 위해 제안된 AGSS (Adaptive Guided Self-Scheduling)[10]는 백-오프 방법을 사용하여 반복을 갖고 오기 위해 경쟁하는 프로세서들의 개수를 줄이고, 각 반복들의 수행 시간이 다양하게 변할 수 있는 경

우를 고려하여 연속된 반복들을 하나의 프로세서에 할당하는 것이 아니라, 연속적인 반복을 서로 다른 프로세서에 각각 흩어지게 할당하므로 부하 불균형의 위험을 작게 하였다. 이러한 예는, 루프 반복의 수행 시간이 선형적으로 감소하는 경우이다. Factoring 기법도 루프 반복의 계산량이 변하는 경우를 반영하기 위한 것으로 루프 할당 단계는 각 배치(batch)별로 이루어지며, 남은 반복의 전부를 고려하는 것이 아니라 그 중 절반의 부분 집합만을 프로세서의 수로 나누어서 할당한다. TSS는 부하 균형을 유지하면서 스케쥴링 오버헤드를 줄이기 위해 제안되었으며, 처음 몇몇 프로세서에게는 큰 덩어리인 $n/2p$ 만큼 할당하고 갈수록 $n/8p^2$ 만큼 크기가 다르도록 연속적으로 작게 할당하는 기법이다.

지금까지 설명한 알고리즘들은 다음과 같은 가정을 전제로 한다. 각 반복들의 작업량이 동일하다면 반복들이 각 프로세서에서 수행되는데 모두 같은 시간이 걸린다. 즉, 프로세서의 성능이 동일함을 의미한다. 그러나, 이러한 가정은 많은 공유 메모리 다중 처리기 환경에서 유효하지 않다. 왜냐하면, 구조상 지역 메모리 혹은 프로세서 캐쉬 메모리의 존재 여부에 따라 달라질 수 있기 때문이다. 이러한 가정을 충족시키기 위해 친화 스케쥴링이 제안되었다. 친화 스케쥴링은 바깥 루프가 순차 루프이고 안쪽 루프가 병렬 구조를 가진 중첩된 루프 구조에 매우 효율적이다. 루프에서의 어떤 반복은 특정한 프로세서에 대해 친화성을 가질 수 있다. 즉, 특정한 프로세서의 지역 메모리나 캐쉬에 특정 반복의 수행에 필요한 자료가 존재할 경우 그 프로세서에 해당하는 반복을 할당한다. 이러한 스케쥴링 기법을 이용하여 분산 환경에서의 가장 비용이 비싼 통신 오버헤드를 감소하여 전체적인 성능을 향상시킬 수 있다[11]. 위에서 살펴본 알고리즘들은 반복들이 독립적인 루프의 할당 기법들이다. 반복들간에 종속 관계가 있는 루프에 대한 효율적인 셀프 스케쥴링에 관한 연구가 필요함에 따라, 다음 3장에서 이러한 루프의 예제와 함께 중앙 큐 기반의 새로운 셀프 스케쥴링 기법을 제안한다. 그리고, 중앙 큐를 사용하는 셀프 스케쥴링 기법인 SS, CSS, GSS, Factoring을 루프 캐리 종속성을 가진 루프 할당에 적용하기 위해 제안된 기법을 이용하여 그들을 동일한 형태로 변형한다.

3. 제안한 할당 기법과 여러 기법들의 변형

3.1 제안한 CDSS 할당 기법

병렬 루프의 스케쥴링은 휴면 프로세서들이 루프의 모든 반복이 끝날 때까지 중앙 작업 큐의 프론트로부터 순서대로 반복을 가져와서 수행한다. 어떤 반복에서 사용된(read/write) 값들이 다른 반복에서 사용 되어지는 반복들간에 종속 관계가 존재하는 루프의 병렬 처리 시에는 고려해야 할 사항이 있다. 그것은 프로세서가 수행을 위해 루프 반복들을 가져와도 그 반복의 종속 관계 만족 여부에 따라 수행 또는 대기 상태가 된다. 즉, 스케쥴링 오버헤드를 줄이기 위해 큰 덩어리를 가져와도 덩어리의 처음 반복의 종속성이 만족 될 때까지 대기 상태가 된다. 이러한 루프 캐리 종속성이 있는 루프의 스케쥴링에 필요한 기본 정의는 자료 종속성(data dependence)이다.

```
Do I = 1 to 20
S1    A[I] = I * 10
S2    B[I] = A[I] + 20
S3    C[I] = D[I-2] * B[I]
S4    D[I] = E[I] + 100
S5    F[I] = (D[I] + C[I]) * B[I]
End Do
```

그림 1 예제 프로그램

그림 1은 루프 캐리 종속성이 발생하는 예제 프로그램이다. 여기서, 루프의 반복 1,2,3,..20이 동시 수행된다고 할 때, 반복 3은 S3문에서 *D[1]*의 자료를 참조(read)하기에 반복 1의 S4 문에서 *D[1]*에 대한 쓰기(write)가 끝나기를 기다리는 동기화가 필요하다.

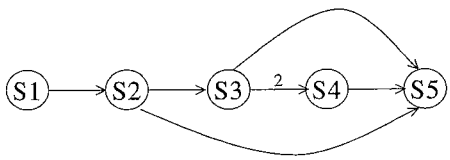


그림 2 작은 성김도의 종속그래프

그림 2는 예제 프로그램의 종속 그래프(dependence graph)로써, 작은 성김도를 위한 반복 내에서 문장들 사이의 종속 관계를 나타내며 그림에서의 숫자는 종속 거리를 나타낸다. 작은 성김도를 위한 병렬성 추출은 문장 S1, S2, S3, S4, S5 각각이 서로 다른 프로세서에서 수행되도록 한다. 루프 스케쥴링은 큰 성김도의 병렬성 추출을 위한 것으로 스케쥴링 단위는 반복내의 문장이 아닌 루프의 반복들이다.

동기화가 필요한 이러한 루프 캐리 종속성이 존재하는 루프를 기존 알고리즘을 이용하여 공유 메모리 다중 처

리기 환경에서 병렬 수행하기 위해서는 알고리즘의 변형이 필요하다. 기존 알고리즘의 변형은 제안한 할당 기법의 알고리즘 일부를 모두 동일하게 적용하여 변형 할 수 있기에 먼저, 본 논문에서 제안하는 새로운 할당 기법에 대해 설명한다. 제안한 할당 기법 CDSS는 알고리즘 1과 같다. 여기서 d 는 종속 거리이며 이를 고려하여 루프를 크게 세 단계로 할당한다. 첫 번째 한 번의 할당이 한 개의 반복을 가지는 초기 단계와 휴면 프로세서가 종속 거리 d 개 크기의 덩어리를 가져오는 중간 단계 그리고, 남은 것이 d 보다 작을 때 전부를 가져오는 마지막 단계로 나눌 수 있다. 본 논문에서 제안하는 CDSS 할당 기법의 수행에 필요한 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 않았다.

알고리즘에 사용된 자료 구조는 중앙 작업 큐, *Cross Dep, Rest, Temp* 변수이다. 중앙 작업 큐는 루프의 모든 반복들의 식별자를 가진다. 첨자가 1부터 시작되는 *CrossDep*은 n(반복 수) 비트의 배열로, 각 원소에 저장된 값은 첨자에 해당하는 반복의 자료 종속성 만족 여부를 나타내는 정보이며 루프의 i번째 반복 L_i의 종속적 후행 반복인 L_j의 수행 가능 여부를 결정한다. *Cross Dep*의 j번째 비트 값이 1이면 L_j의 선행 노드인 L_i의 수행이 끝나 자료 종속성이 만족되어 L_j가 현재 수행 가

알고리즘 1 제안한 CDSS 할당 기법

```
Entry Block :
    /* get_routine */
1   lock(Queue)
2       Temp = Queue
3   unlock(Queue)
4   if (Temp[front] == 1) then
5       i = get_from_queue(1)                  /* 루프의 시작 부분 할당단계 (큐의 프론트로부터 한개 할당) */
6       Execute Block(1)
7   else  if (Rest > d-1)
8       start_chunk = get_from_queue(d)        /* 루프의 중간 부분 할당단계 (front에서 d개씩 가져옴) */
9   else
10      start_chunk = get_from_queue(Rest)     /* 루프의 마지막 부분 할당단계 (남은 것 모두 할당) */
11  end if
12  update Rest                                /* Rest 값을 갱신 */

    /* exec_test_routine */
13  for k=0, ChunkSize - 1
14    i = start_chunk + k
15    lock(CrossDep)
16      Temp = CrossDep
17    unlock(CrossDep)
Exec_test:
18    if (Temp[i] == 1) then
19      Execute Block(i)
20    else
21      waiting for system_defined interval
22      reload Temp from CrossDep
23      Exec_test
24    end if
25  end for

Execute Block :
26    executable code

Exit Block :                                   /* fetch_set_routine */
27  j = detect(i)                              /* j = i+d  */
28  lock(CrossDep)
29      fetch_set(j, 1)
30  unlock(CrossDep)
```

능함을 의미한다. 초기 값으로, 종속 그래프에서 들어오는 종속 아크가 없는 반복들은 1로 셋(set) 되어진다. 즉, 이러한 반복들은 독립적이기에 바로 수행 할 수 있으며 나머지는 모두 0으로 채워진다. 임계 영역에 해당하는 공유 변수 *CrossDep*와 중앙 작업 큐는 잠금(lock)으로 직렬화(serialization)를 유지하며, 지역 변수로 사용된 *Temp*는 공유 메모리에 접근 해 있는 시간을 줄이기 위해 사용된다. *Rest* 변수는 현재 남은 반복의 수를 나타낸다. 알고리즘은 크게 세 부분, 알고리즘 1의 1-25번 줄의 입구 부분과 26번 줄의 수행 부분 그리고, 27-30번 줄의 출구 부분으로 나누어진다. 입구 부분은 스케줄링을 위해 추가되어지며 1-12번 줄의 *get_routine*에서는 반복의 수행을 위해 작업 큐의 프론트로부터 적당한 개수의 루프 반복을 가져온다. 가져오는 반복 개수에 따라 스케줄링 횟수가 결정되며 이것은 실행 시간 오버헤드와 밀접한 관계가 있다. 1-3번 줄은 임계 영역에 접근 해 있는 시간을 줄이기 위해 임시 기억 장소에 큐의 내용을 저장한다. 제안한 할당 기법에서 가져오는 덩어리의 크기는 루프의 시작, 중간, 끝 부분이 각각 틀리다. 4-6번 줄은 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. 7-8번 줄은 루프의 중간 부분 할당 단계로, 현재 남은 반복의 수(*Rest*)가 d 보다 크거나 같을 때 큐로부터 d 만큼의 반복들을 가져온 후 덩어리의 첫 번째 반복 식별자를 *start_chunk*에 리턴 한다. 9-10번 줄은 루프의 마지막 부분 할당 단계로, 남은 것이 d 보다 작을 때 남은 것 모두를 가져온다. 12번 줄은 가져온 덩어리를 현재 남은 반복의 수에 적용하기 위해 *Rest* 변수를 갱신한다.

3.2 기존 할당 기법의 변형

중앙 큐 기반의 셀프 스케줄링 알고리즘들을 이용하여 루프 캐리 종속성을 가진 루프를 수행하기 위해서는 기존 알고리즘들의 변형이 필요하다. 각 프로세서가 수행을 위해 가져온 반복들의 종속성 만족 여부를 체크하는 루틴이 추가 되어야하며, 이러한 연산 결과에 따라 가져온 반복이 수행 또는 대기 상태가 된다. 그리고 만약, 수행 가능하다면 수행 후 그것에 종속적인 후행 반복의 종속 정보를 갱신하는 루틴 또한 필요하다. 지금까지 설명한 알고리즘의 변형이 필요한 부분은 제안한 CDSS 할당 기법(알고리즘 1)의 13-30번 줄을 기존 알고리즘에 동일하게 적용시키면 되며, 종속성 만족 여부를 표현하기 위해 사용된 *CrossDep* 자료 구조가 필요하다.

알고리즘 1의 13-25번에 해당하는 *exec_test_routine*에서는 가져온 덩어리의 첫 번째 반복부터 현재 수행 가능한지를 검사하는 루틴으로 종속 관계가 만족되었을 때 실행을 하게 된다. 13번 줄의 *ChunkSize*는 각 스케줄링 정책별로 가져오는 덩어리의 크기로 정책에 따라 다르다. 알고리즘에서 *CrossDep*를 재 적재하여 실행 가능 여부를 체크하는 주기는 실행 시간 오버헤드와 관계되어 루프 수행 시간에 영향을 주게된다. 임계 영역에 접근해 있는 시간을 줄이기 위해 *CrossDep* 내용을 임시 변수에 저장하며, i 번째 비트가 1이면 i 번째 반복의 종속성이 만족되었기에 반복을 수행한다. 만약, 종속성이 만족되지 않은 경우는 대기 상태가 되어 시스템이 지정한 간격 동안 대기하며 갱신된 *CrossDep*에 다시 접근하여 검사 과정을 반복한다. 수행 부분은 루프 몸체의 코드를 수행한다. 출구 부분의 *fetch_set* 루틴은 현재 수행이 끝난 반복을 *CrossDep*에 반영하기 위해 반복 L_i의 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.

4. 루프 할당 기법의 비교 분석

루프 캐리 종속성을 가진 루프를 병렬 처리하기 위해 본 논문에서 제안한 CDSS와 변형된 SS, CSS, GSS, Factoring의 네 개 알고리즘을 수행 시간 스탭, 부하 균형도, 동기화에 따른 지연 시간, 프로세서 병렬성을 시뮬레이션 하였다. 시뮬레이션을 위해 먼저 각 할당 기법의 스케줄링 방법을 알아본다.

할당 기법 SS를 이용하여 루프를 스케줄링 할 경우, 각 프로세서는 항상 하나의 반복만을 가져오기에 스케줄링 횟수는 전체 반복의 개수 만큼이다. CSS 기법에서는 휴면 프로세서들이 동일한 크기의 덩어리(k개 반복)를 할당받는다. 적당한 k 값을 선택하기가 힘들며 일반적으로, k 는 $\lceil n/p \rceil$ 로 결정된다. 덩어리 크기를 동적으로 다르게 변화시키는 GSS의 덩어리 크기 결정은 다음과 같이 이루어진다.

$$R_0 = n,\ R_{i+1} = R_i - G_i,\ G_i = \lceil R_i/p \rceil$$

여기서, R_i는 i 번째 스케줄링 단계의 남은 반복의 수이며 G_i는 i 번째 스케줄링에 할당되는 덩어리의 크기이다. Factoring 기법의 루프 할당은 각 배치별로 이루어지며 GSS 할당 기법과 같이 남은 반복의 전부를

고려하는 것이 아니라, 그 중 절반의 부분 집합만을 프로세서의 수로 나누어서 할당한다. 각 배치는 동일한 크기의 덩어리를 p 번 아래와 같이 할당한다.

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

여기서, F_i는 i 번째 배치의 스케줄링에 사용된 덩어리의 크기이다. 일반적으로, x_i를 최적의 값인 2[6]로 하여 스케줄링 한다.

시뮬레이션에 사용된 응용 프로그램의 파라메터 값으로 루프 반복의 수(n)가 20, 32, 60인 루프에 대해 자료 종속성의 종속 거리(d)를 각각 2, 3, 4로 변화하였다. $L(i)$는 i 번째 반복의 수행 시간(계산량)으로 모든 반복에 대해 동일하게 단위 시간으로 1로 가정한다. 시스템 파라메터인 사용된 프로세서 수(p)는 최대 4로 하였으며, S_{one}는 작업 큐에 대한 동기화(접근 시 잠금과 해제)를 포함한 한 번의 스케줄링 연산 시간으로 시스템마다 사용하는 동기화 프리미티 오퍼레이션에 따라 비용이 다양하기에 $L(i)$ 보다 작다고 가정하여 단위 시간 0, 0.1, 0.5로 설정하였다. 전제 조건으로, 루프 수행에 참가할 p 개의 프로세서가 있다면 처음 스케줄링은 $P_1, P_2, P_3, \cdots, P_p$ 순으로 큐에 접근하여 각 할당 정책에 따라 정해진 개수의 루프 반복을 할당받고 이후는 휴면 상태가 된 프로세서가 순서대로 스케줄링에 참여하며, 모든 프로세서는 동일한 성능을 가진다.

4.1 루프의 수행 시간

루프 수행 시간에 영향을 미치는 스케줄링 오버헤드, 응용의 종속 거리, 프로세서 수에 따른 각 정책별 수행 시간 스텝을 비교 하고자 한다. 알고리즘 1의 1-12번 줄의 루프 할당 부분만이 각 스케줄링 정책별로 다르다. 13-25번 줄의 반복의 수행 가능 여부를 체크하는 루틴과 27-30번 줄의 출구 부분은 변형된 할당 기법에서 모두 동일하게 사용되기에 루프의 수행 시간 측정에서 제외하였다. 루프 수행 시간을 결정하는 요소인 S_{count}는 루프 수행을 위한 스케줄링 횟수이며, S_{total}는 전체 스케줄링 오버헤드($S_{total} = S_{count} * S_{one}$)이다.

4.1.1 스케줄링 오버헤드

1) 스케줄링 오버헤드를 고려하지 않은 경우($S_{one}=0$)

표 1은 스케줄링 오버헤드를 고려하지 않았을 경우($S_{one}=0$)의 할당 정책별 각 반복이 수행되는 시각과 각 프로세서가 수행한 반복들을 나타낸 것으로 실험 결과 중 $d=2, p=4, n=60$ 인 경우이다. 표에서 첫 번째 칸의 진한 숫자는 수행 시간 스텝을 나타내며 음영으로 표시된 부분들은 한 번 스케줄링시 할당되는 덩어리 크기를 의미한다. 스케줄링 오버헤드를 고려하지 않은 루프의 수행 시간(L_{exec})은 각 반복이 자료 종속성을 만족시키며 수행 가능한 최소 시간이다. 표의 CSS 정책에서 P_2가 가져온 덩어리의 처음 반복 16은 P_1이 할당받은 14번 반복의 수행이 끝난 후 수행 가능하다. CSS 정책에 따른 루프의 수행 시간은 57이며, 스케줄링 횟수(S_{count})는 사용된 프로세서 개수인 4이다. Factoring은 4개 배치가 사용되고 각 배치별 덩어리 크기는 8, 4, 2, 1 이며 L_{exec}는 47, S_{count}는 16(배치 수 * p)이다. 표에서 변형된 SS 기법이 60번의 스케줄링 횟수로 가장 많은 스케줄링 오버헤드를 유발하지만 31번의 할당을 하는 제안한 CDSS와 함께 수행 시간이 가장 짧다. 스케줄링 오버헤드를 고려하지 않은 경우는 다양한 파라메터 값(p,n,d)에 대해 평균적으로 변형된 CSS, GSS, Factoring, SS 그리

표 1 스케줄링 오버헤드를 제외한 경우의 반복 수행 시각 ($d=2, p=4, n=60$, $L(i)=1$, $S_{one}=0$)

schedule style		time step
CSS	p1	1 2 3 4 5 6 7 8 9 0 1 2 3 4 5
	p2	6 7 8 9 0 1 2 3 4 5 6 7 8 9 0
	p3	1 2 3 4 5 6 7 8 9 0 1 2 3 4 5
	p4	6 7 8 9 0 1 2 3 4 5 6 7 8 9 0
GSS	p1	1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 3 4 5 6 7 7
	p2	6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 8
	p3	8 9 0 1 2 3 4 5 6 2 3 4 9
	p4	7 8 9 0 1 2 5 6 0
Fact.	p1	1 2 3 4 5 6 7 8 3 4 5 6 9 0 7
	p2	9 0 1 2 3 4 5 6 7 8 9 0 1 2 8
	p3	7 8 9 0 1 2 3 4 1 2 3 4 3 4 9
	p4	5 6 7 8 9 0 1 2 5 6 7 8 5 6 0
SS	p1	1 5 9 3 7 1 5 9 3 7 1 5 9 3 7
	p2	2 6 0 4 8 2 6 0 4 8 2 6 0 4 8
	p3	3 7 1 5 9 3 7 1 5 9 3 7 1 5 9
	p4	4 8 2 6 0 4 8 2 6 0 4 8 2 6 0
CDSS	p1	1 8 9 6 7 4 5 2 3 0 1 8 9 6 7
	p2	2 3 0 1 8 9 6 7 4 5 2 3 0 1 8 9
	p3	4 5 2 3 0 1 8 9 6 7 4 5 2 3 0
	p4	6 7 4 5 2 3 0 1 8 9 6 7 4 5

표 2 스케쥴링 오버헤드를 포함한 성능 비교 (d=3, $L(i)$=1, S_{one}=0.1,0.5)

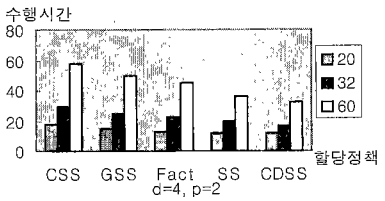
p	n	schedule style																			
		CSS				GSS				Factoring				SS				CDSS			
		L_{exec}	S_{count}	L_{total}		L_{exec}	S_{count}	L_{total}		L_{exec}	S_{count}	L_{total}		L_{exec}	S_{count}	L_{total}		L_{exec}	S_{count}	L_{total}	
4	20	14	4	14.4	16	11	9	11.9	15.5	8	12	9.2	14	7	20	9	17	8	8	8.8	12
	32	26	4	26.4	28	20	10	21	25	16	16	17.6	24	11	32	14.2	27	12	12	13.2	18
	60	54	4	54.4	56	44	12	45.2	50	39	16	40.6	47	20	60	26	50	21	21	23.1	31.5
3	20	16	3	16.3	17.5	12	7	12.7	15.5	11	8	11.8	15	7	20	9	17	8	8	8.8	12
	32	26	3	26.3	27.5	22	8	22.8	26	18	11	19.1	23.5	11	32	14.2	27	12	12	13.2	18
	60	56	3	56.3	57.5	47	9	47.9	51.5	40	15	41.5	47.5	20	60	26	50	21	21	23.1	31.5
2	20	18	2	18.2	19	15	5	15.5	17.5	13	8	13.8	17	10	20	12	20	10	8	10.8	14
	32	30	2	30.2	31	25	6	25.6	28	22	10	23	27	16	32	19.2	32	16	12	17.2	22
	60	58	2	58.2	59	52	6	52.6	55	47	10	48	52	30	60	36	60	30	21	32.1	40.5

고 제안한 CDSS가 각각 33.6, 28.1, 24.4, 15.8, 16.3의 수행 시간 스텝을 유지하였으며, SS 할당 기법이 가장 좋은 성능을 보였다. 제안한CDSS는 CSS, GSS, Factoring, SS에 대해 각각 51, 42, 33, -3%의 성능 향상을 가졌으며, SS, CDSS, Factoring, GSS, CSS 기법 순으로 성능이 우수하였다.

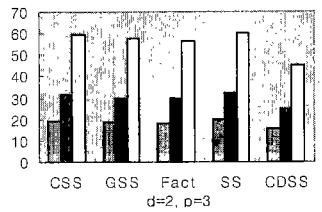
2) 스케쥴링 오버헤드를 고려한 경우(S_{one}=0.1,0.5)

스케쥴링 연산 시간을 고려하여 오버헤드가 큰 경우(S_{one}=0.5)와 작은 경우(S_{one}=0.1)에 대해 시뮬레이션 하였다. 스케쥴링 오버헤드를 고려한 실제 루프의 수행 시간(L_{total})은 L_{exec}와 할당 정책에 따른 S_{total}의 합으로 측정된다. 표 2는 스케쥴링 오버헤드를 고려하여 루프 할당 기법들의 성능을 시뮬레이션 한 결과 중에 종속 거리가 3인 경우이다. 할당 기법별로, 표의 첫 번째 열은 자료 종속 관계를 만족시키며 병렬 수행 한 L_{exec}이며, 두 번째 열은 S_{count}를 나타낸다. S_{total}은 S_{count}와 S_{one}의 곱으로 표현된다. 세 번째, 네 번째 열은 각각 S_{one}이 0.1, 0.5일 때의 S_{total}과 L_{exec}을 합한 L_{total}이다.

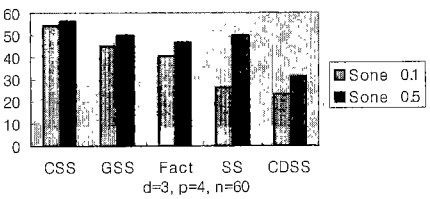
그림 3은 스케쥴링 연산을 고려한 경우의 반복 수에 따른 각 정책별 수행 시간을 나타낸 것이다. 한 번의 스케쥴링 연산 시간이 반복 수행 시간에 비해 매우 짧은 경우(S_{one}= 0.1)는 종속 거리, 프로세서 수, 반복 수의 변화에 관계없이 제안한 CDSS가 가장 성능이 우수하였으며 SS, Factoring, GSS, CSS 할당 기법 순으로 좋은 성능을 보였다. 제안한 CDSS는 CSS, GSS, Factoring, SS 기법에 비해 평균적으로 각각 47, 38, 30, 8 %의 성능 향상을 보였다. SS 할당 기법은 많은 스케쥴링 오버헤드를 유발하지만 스케쥴링 오버헤드가 반복 수행 시간에 비해 매우 작다면 다른 할당 기법에 비해 좋은 성능을 얻을 수 있다(그림 3(a)). 반면에, 한



(a) S_{one}=0.1



(b) S_{one}=0.5



(c) S_{one}의 변화

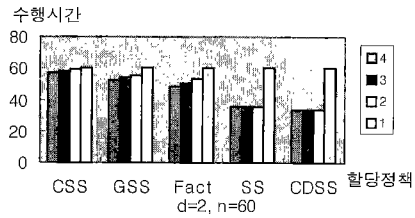
그림 3 스케쥴링 연산시간에 따른 수행 시간

번의 스케쥴링 연산 시간이 큰 경우(S_{one}=0.5)에 대해 시뮬레이션 한 결과, 종속 거리가 2인 응용에서는 CDSS가 가장 좋은 성능을 보이며, Fatoring, GSS, CSS, SS 기법 순으로 성능이 우수하였다(그림 3(b)). 종속 거리가 3, 4인 루프에 대해서는 CDSS, Factoring, GSS, SS, CSS 할당 기법 순으로 좋은 성능을 보여 제

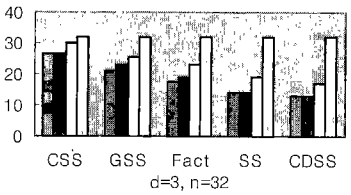
안한 할당 기법이 모든 경우에 대해 가장 적은 수행 시간을 유지함을 알 수 있었다. 스케쥴링 오버헤드가 큰 경우는 SS 할당 기법이 좋은 성능을 보이지 못했고 Factoring이 다른 기법들에 비해 우수하였다. S_{total}는 CSS가 가장 작았으며, GSS, Factoring, CDSS, SS 기법 순으로 작았다. 그림 3(c)는 S_{one} 값에 따른 수행 시간을 나타낸다.

4.1.2 종속 거리

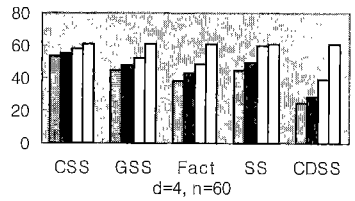
스케쥴링 연산 시간을 고려하여 종속 거리별 프로세서 수에 따른 루프 수행 시간을 알아보고자 한다. 그림 4에서 보듯이 종속 거리에 따라 할당 정책들간의 성능의 우수성이 조금씩 달랐다. 대부분의 경우, 종속 거리에 관계없이 병렬 처리를 위해 많은 수의 프로세서를 투입할수록 루프 수행 시간을 줄여 효율적임을 알 수 있었다. 하지만, 그렇지 않은 경우도 있었다. S_{one}이 0.1이라 할 때, 종속 거리가 2인 루프에 대한 시뮬레이션 결과인 그림 4(a)에서 다른 할당 기법들은 프로세서 개수를 많이 사용할수록 효율적임에 반해 SS와 CDSS는 2개의 프로세서를 투입했을 때 가장 효율적임을 알 수 있다. 그림



(a) d=2, S_{one}=0.1



(b) d=3, S_{one}=0.1



(c) d=4, S_{one}=0.5

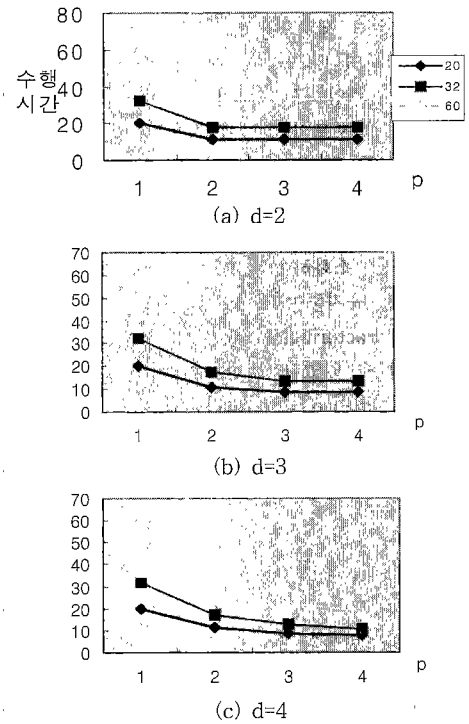
그림 4 종속 거리별 수행 시간

4(b)는 종속 거리가 3인 루프의 실험 결과로써 2개 보다 3개의 프로세서를 사용했을 때 수행 시간을 줄일 수 있음을 보이며, CSS, SS 그리고 제안된 CDSS 할당 기법들에서는 4개의 프로세서를 사용해도 성능 향상에 도움을 주지 못하는 반면에, 할당 기법 GSS, Fatoring은 많은 수의 프로세서를 사용할수록 수행 시간을 감소시킬 수 있었다. 종속 거리가 4인 경우의 그림 4(c)에서 모든 할당 기법에서 4개의 프로세서를 사용할 때 가장 효율적임을 알 수 있으며, SS 할당 기법은 2개의 프로세서를 투입해도 단일 프로세서와 거의 같은 성능을 보인다. 여러 할당 기법 중 SS는 파라메터 값의 변화에 따라 성능의 우수성 순위가 자주 바뀌었다. 그림 4(a),(b)에서 보듯이 SS 기법이 CDSS 외의 다른 기법에 비해 우수하나 항상 그런 것은 아니다(그림 4(c)).

스케쥴링 연산 시간을 포함하여 다양한 반복 수, 프로세서 수에 대해 종속 거리가 2인 응용에서 제안한 할당 기법은 CSS, GSS, Factoring, SS에 비해 평균적으로 각각 32, 27, 24, 18%, 종속 거리가 3인 응용에서는 42, 35, 28, 24% 그리고, 종속 거리가 4일 때는 45, 35, 26, 27%의 성능 향상을 보였다. 대체적으로 CDSS, SS, Factoring, GSS, CSS 순으로 성능이 우수함을 알 수 있었고, 종속 거리가 클수록 본 논문에서 제안한 할당 기법이 다른 알고리즘에 비해 더욱 효율적이었다.

4.1.3 프로세서 수

그림 5는 제안한 CDSS 기법에서의 프로세서 수에 따른 수행 시간이다. 그림 5(a)의 종속 거리가 2인 경우는 다양한 반복 수에 관계없이 SS를 제외한 다른 할당 기법들은 프로세서 개수를 많이 사용할수록 효율적임에 반해 CDSS는 2개의 프로세서를 투입했을 때 최적임을 알 수 있다. 즉, 사용된 프로세서 수가 2, 3, 4 개일 때 동일한 성능을 얻었다. 종속 거리가 3인 경우의 그림 5(b)에서 프로세서를 2개 보다 3개를 사용했을 때 수행 시간이 줄어들며 3개 이상의 프로세서를 투입해도 수행 시간이 동일함을 알 수 있다. 그림 5(c)는 종속 거리가 4인 경우로 4개의 프로세서를 사용했을 때 다양한 루프 크기에 관계없이 최소의 수행 시간을 가진다. 본 논문에서 제안한 CDSS 할당 기법으로 루프 캐리 종속성이 존재하는 루프의 병렬 수행을 위해 사용되어지는 프로세서의 수는 반복의 수에 상관없이 효율성 측면에서 종속 거리와 밀접한 관계가 있음을 알 수 있었다. 즉, 종속 거리가 d 인 루프의 수행 시, d 개 이상의 프로세서를 투입해도 성능 향상에 도움을 주지 못하며 프로세서를 d 개 사용하는 것이 최적임을 시뮬레이션을 통해 알 수 있었다.



(a) d=2

(b) d=3

(c) d=4

그림 5 프로세서 수에 따른 수행 시간(S_{one}=0.1)

지금까지 루프 수행 시간에 영향을 주는 요소들에 대해 살펴보았다. 다양한 파라메터 값에 대해 제안한 CDSS 할당 기법이 모든 경우에 최소의 수행 시간을 유지하였으며, 다른 할당 기법들은 d, p, n, S_{one}에 따라 성능이 조금씩 차이가 났다. S_{one}이 0.1, 0.5인 두 경우를 모두 고려했을 때, 제안한 CDSS 기법은 SS, Factoring, GSS, CSS에 대해 각각 23, 26, 32, 39%의 성능 향상을 보였다. 스케쥴링 오버헤드를 고려하지 않은 경우(S_{one}=0)도 포함할 경우 평균적으로 CDSS, SS, Factoring, GSS, CSS 순으로 성능이 우수하였으며, CSS 기법이 가장 성능이 떨어졌다.

4.2 부하 균형

최적의 동적 루프 스케쥴링 알고리즘의 조건은 부하 균형을 유지함으로 좋은 프로세서 성능을 얻고, 스케쥴링 오버헤드를 줄여 수행 시간을 최소함으로 효율성을 높이는 것이다. 이러한 조건들은 서로 상충 관계가 있기에 모두를 만족시키기는 불가능하다. 부하 균형은 할당되는 덩어리의 크기가 작을 때 좋지만 많은 스케쥴링 오버헤드를 필요로 한다. 표 3은 반복 수와 프로세수 개수에 따른 프로세서 각각이 수행한 반복의 수를 나타내는 부하 균형도의 시뮬레이션 결과로 종속 거리가 2인 경우이다. 부하 균형은 프로세서별 작업에 참여한 시간이 동일함을 의미하며 프로세서별 작업한 반복들의 수가 다르다고 해서 부하 불균형이라고는 볼 수 없다. 왜냐하면, 루프 특성에 따라 각 반복들의 수행 시간이 선형적으로 감소 혹은 증가하는 경우는 프로세서 당 작업한 반복들의 수와 프로세서가 작업에 참여한 시간간의 비례 관계가 성립되지 않기 때문이다. 다양한 파라메터 값에 대해 CSS, SS, Factoring 기법은 거의 완벽한 부하 균형을 보이는 반면에, GSS는 부하 불균형을 보였다. 제안한 CDSS는 CSS, SS, Factoring 기법처럼 완전하지는 않지만 양호한 부하 균형을 보였다.

4.3 지연 시간

시뮬레이션을 통해 루프 수행 시간에 영향을 주는 또다른 요소를 찾을 수 있었다. 그것들에 대해 알아본다. $d_{start(i)}$는 프로세서 i가 최초로 반복을 할당 후 반복의 수 수행을 처음 시작하기까지의 프로세서 i의 시작 지연시간이며 d_{start}는 프로세서의 시작 지연 시간($d_{start}=\sum_{i=1}^{p} d_{start(i)}$)이다. $d_{chunk(i)}$는 프로세서 i가 할당받은 각 청크의 마지막 반복을 수행 후, 그 다음 청크의 처음 반복을 수행하기까지의 모든 청크에 대한 지연 시간 합을 나타내는 프로세서 i의 청크 지연 시간이며 d_{chunk} 는 프로세서의 청크 지연 시간 ($d_{chunk}=\sum_{i=1}^{p} d_{chunk(i)}$)이다. d_{total}는 전체 스케쥴링 오버헤드와 종속성 만족을 위한 동기화에 따른 프로세서의 시작, 청크 지연 시간을 포함한 전체 지연 시간($d_{total}= S_{total}+ d_{start}+ d_{chunk}$)이다. 예로서, 표 1의 GSS 정책에서 $d_{start(i)}$는 P_1, P_2, P_3, P_4 순으로 각각 0, 14, 25, 33 이며 d_{start}는 72이다. $d_{chunk(i)}$는 P_1은 23, 5로 28이며, P_2, P_3, P_4는 각각 19, 12, 9로 d_{chunk}는 68이다. 표 4는 S_{one}이 0.1일 때의 그림 1 예제 루프의 수행에 따른 각 정책별 지연 시간들을 나타낸 것이다. 다양한

표 3 프로세서별 수행한 반복 수(d=2)

d	p	id	n = 20					n = 32				
			CSS	GSS	Fact	SS	CDSS	CSS	GSS	Fact	SS	CDSS
2	4	p1	5	8	5	5	5	8	12	8	8	8
		p2	5	5	5	5	6	8	9	8	8	8
		p3	5	4	5	5	5	8	6	8	8	8
		p4	5	3	5	5	4	8	5	8	8	8
	3	p1	7	10	7	7	7	11	15	11	11	11
		p2	7	6	7	7	7	11	10	11	11	11
		p3	6	4	6	6	6	10	7	10	10	10
	2	p1	10	14	10	10	10	16	21	16	16	16
		p2	10	6	10	10	10	16	11	16	16	16

표 4 지연 시간 (d=2,n=20,p=4, S_{one}=0.1)

delay \ style	CSS	GSS	Fact	SS	CDSS
S_{total}	0.4	0.9	1.2	2	1.1
d_{start}	24	20	12	2	3
d_{chunk}	0	12	15	16	14
d_{total}	24.4	32.9	28.2	20	18.1

파라메터 값에 대해 d_{total}은 CDSS 기법이 가장 작으며 SS, CSS, Factoring, GSS 순으로 작았다. 루프 수행 시간 또한 CDSS가 가장 짧으며 SS, Factoring, GSS, CSS 순으로 적게 걸렸다.

4.4 프로세서 병렬성

루프 수행 시간이 적게 걸리는 할당 기법은 다른 알고리즘에 비해 상대적으로 지연 시간 또한 적어야 하지만, 표 4에서 보듯이 GSS가 전체 지연 시간이 가장 큼에도 불구하고 CSS 보다 수행 시간이 적게 걸렸다. 그 이유는 수행 시간을 결정하는 또 다른 요소가 존재하기 때문이다. 그것은 같은 시각에 동시 수행되는 작업량을 나타내는 프로세서 병렬성이다. 프로세서 병렬성의 예로, 표 1의 CSS 기법에서 프로세서 P_1, P_2가 반복 15, 16을 동시 수행하듯이, CSS, GSS, Factoring, SS, CDSS 기법이 각각 3, 9, 13, 30, 30번씩 d 개(여기선, 2개)의 서로 다른 프로세서에서 같은 시각에 반복들을 하나씩 병렬 수행한다. 여기서, SS 기법은 좋은 프로세서 병렬성을 제공하지만 다른 기법에 비해 많은 스케쥴링 오버헤드를 가지는 것이 단점이다. 할당 기법 SS와 CDSS가 동일한 프로세서 병렬성을 가지지만, SS 기법은 많은 스케쥴링 오버헤드 때문에 CDSS에 비해 수행 시간이 길었다. 파라메터 값을 변화시켜 시뮬레이션 한 결과, 할당 기법들에 따라 프로세서 병렬성의 정도가 달랐으며 그것은 수행 시간에 큰 영향을 주었다. 제안한 CDSS와 SS 기법이 가장 많은 프로세서 병렬성을 제공하였으며, Factoring, GSS, CSS 순으로 병렬성이 좋았으며 수행 시간 또한 위의 순서대로 짧아짐을 알 수 있었다. 가장 효율성이 좋지 못한 CSS는 프로세서 병렬성 또한 가장 좋지 못했다. 그리고, 반복수와 프로세서 수에 상관없이 모든 할당 기법들이 동시 수행 가능한 프로세서들의 수 혹은 반복의 수는 최대 d 개임을 알 수 있었다.

5. 결 론

본 논문에서는 루프 캐리 종속성을 가진 루프의 효율적 수행을 위한 새로운 중앙 큐 기반의 셀프 스케쥴링 기법을 제안하였다. 그리고, 제안된 CDSS 기법을 이용하여 기존의 중앙 큐 기반의 셀프 스케쥴링 기법들을 종속성을 가진 루프의 할당에 적용될 수 있음을 보였으며, 그들의 성능을 비교 분석하였다. 응용 및 시스템 파라메터의 다양한 실험 값에 대해 시뮬레이션 한 결과, 제안된 할당 기법은 양호한 부하 균형을 유지하며 전체 지연 시간을 최소화하여 다른 할당 기법에 비해 최소의 수행 시간을 유지하여 효율적임을 보였다. 평균적으로 CDSS, SS, Factoring, GSS, CSS 기법 순으로 성능이 우수함을 알 수 있었다. 또한, 종속 거리가 큰 응용일수록 제안된 기법이 다른 알고리즘에 비해 매우 좋은 성능을 보였으며 높은 프로세서 병렬성을 제공하였다. CDSS 할당 기법에서는 종속 거리에 관계하여 d 개의 프로세서를 사용했을 때 효율적임을 알 수 있었다. 즉, 종속 거리가 d 인 루프 수행을 위해 d 개 이상의 프로세서를 투입해도 성능 향상에 도움을 주지 못하고, 한번에 동시 수행이 가능한 최대 프로세서 수는 d 개임을 알 수 있었다. 반복의 수가 많고 반복의 수행 시간이 짧은 응용에서는 중앙 작업 큐의 빈번한 접근으로 병목 현상이 생긴다. 이것을 줄이기 위해 *CrossDep*를 여러 개로 분배하면 효율성을 높일 수 있다고 예상된다. 향후 연구는 단일 처리기를 포함한 여러 플랫폼에서 제안된 알고리즘과 변형된 기법들의 성능 평가를 위해 Java 스레드(thread)를 이용한 효율적인 구현 방안에 대해 고찰하고자 한다.

참 고 문 헌

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

[2] H. Zima and B .Chapman, *Super Compiler for Parallel and Vector Computers*, Addison-Wesley, 1991

[3] C. P. Kruskaland and A. Weiss, "Allocating independent subtasks on parallel processors," *IEEE Trans. Software Eng.*, vol. 11, no. 10, pp. 1001-1016, Oct, 1985

[4] 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

[5] 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

[6] 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

[7] 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

[8] 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

[9] J. Liu, V. A. Saletore, and T. G. Lewis, "Safe Self-Scheduling: A Parallel Loop Scheduling Scheme for Shared-Memory Multiprocessors," *Int. Parallel Programming*, vol. 22, no. 6, pp.589-616, 1994

[10] 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

[11] 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

김 현 철

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

유 기 영

1976년 경북대학교 수학교육학과 졸업(이학사). 1978년 한국과학기술원 전산학과 졸업(공학석사). 1992년 미국 Rensselaer Polytechnic Institute 졸업(이학박사). 1978년 ~ 현재 경북대학교 컴퓨터공학과에 재직. 관심분야는 병렬처리, DSP array processor 설계, 암호화 등