공유메모리 시스템에서 개선된 병렬 루프 스케쥴링 알고리즘

이영규, 박두순
° 극동정보대학 전산정보처리과
°° 순천향대학교 정보기술공학부

Improved Parallel Loop Scheduling Algorithm on Shared Memory Systems

Young-Kyu Lee, Doo-Soon Park
° Dept. of Computer Information Process, Keukdong College
°° Division of Computer Science and Engineering, Soonchunhyang University

요 약

병렬 시스템 환경에서 최적의 스케쥴링을 수행하기 위해서는 병렬성을 가진 iteration들에 대해 최소의 동기화 오버헤드와 load balance가 달성하도록 스케쥴링을 수행해야 한다. 다중 프로세서들은 실행을 위하여 메모리로부터 iteration들에 대한 chunk를 계산한후 할당받게 된다. 이때, 각 프로세서들의 상호 배타적인 메모리 접근으로 많은 오버헤드 및 병목현상이 발생된다. 또한, 프로세서에게 할당된 chunk내 iteration들의 실행시간 분포가 서로 상이한 경우에는 load imbalance의 원인이 되어 결과적으로 전체 스케쥴링에 나쁜 영향을 준다. 따라서, 최적의 스케쥴링을 수행하기 위해서 본 논문에서는 기존의 스케쥴링 방법들에서 문제점들을 도출하고, 자료의 국부성과 프로세서 동족성을 고려한 개선된 병렬 루프 알고리즘을 제안하고, 성능평가를 통해 개선된 알고리즘이라는 것을 보였다.

1. 서론

병렬 시스템 환경에서 프로세서들을 독립적으로 충분히 활용하여 최대의 병렬성을 가진 응용프로그램을 효과적으로 실행하는 것이 병렬 스케쥴링이다. 병렬성을 가진 모든 iteration들은 처리 순서에 무관하게 동시에 독립적으로 실행될 수 있으며, 이때, 모든 iteration들의 분포(distribution: μ, σ)가 균일한 경우에는 서로 독립된 iteration들을 컴파일 시간에 프로세서에게 균등하게 할당하는 정적 스케쥴링이 효과적일 수 있다. 그러나 각각의 iteration들이 서로 불규칙적인 경우에는 load imbalance가 발생되어 정적 스케쥴링은 효과적일 수 없다. 불규칙 iteration들을 효과적으로 수행하기 위해서는 수행 시간에 idle 프로세서에게 할당되는 동적 스케쥴링이 필요하며, 약간의 오버헤드가 수반된다. 따라서, 효과적인 스케쥴링을 수행하기 위해서는 동적 스케쥴링과 정적 스케쥴링이 모두 고려되어야한다. 최적의 스케쥴링을 위해서 고려되어야할 몇몇 요소들을 살펴보면 다음과 같다. 첫째, 각 iteration들의 분포를 고려해야 한다. 그러나 각 iteration들의 분포가 컴파일 시간에 조차 알려지지 않기 때문에 기존의 스케쥴링 알고리즘에서는 각 iteration들을 서로 임의의 독립된 변수로 취급하여 처리하고 있다[2,4,6]. 둘째, 데이타 국부성을 고려해야한다. 이는 서로 인접한 iteration들이 가장 근사한 분포를 가진다는 확률적 의미를 반영하는 것이다[5]. 세째, 각 타스크를 프로세서에게 할당할 때, 소요되는 전송시간이 고려되어야 하며, 이는 네트워크의 구성 형태와 관련이 있다. 네째, 각 프로세서들의 성능이 고려되어야 하며, 이종 시스템에서 특히 반영 될 요소라 할 수 있다. 본 논문에서는 프로그램 병렬성의 근원이 되는 루프에서 크기나 타스크의 분포가 입력 데이터에 의존하여 병렬 수행을 하는 불규칙적인 루프에 초점을 둔다. 이런 불규칙적인 루프를 효과적으로 수행하기 위해서 본 논문에서는 컴파일 시간에 iteration들에 대한 자료 국부성을 고려하여 제어키에 iteration들을 여러개의 그룹으로 분해하고, 분해된 그룹들을 프로세서 동족성을 고려하여 정적으로 할당하므로서 스케쥴링 오버헤드와 load balance 사이의 tradeoff 관계를 최소화 하였다.

본 연구는 정보통신진흥원의 '99년 대학 기초연구 지원비에 의한 것임.

2. 이론적 배경

스케쥴링 방법은 크게 전역 프로그램 정보인 매개변수를

토대로 컴파일 시간에 병렬 타스크들을 균등하게 할당하므로서 낮은 스케쥴링 오버헤드를 달성할 수 있는 정적 스케쥴링과 load imbalance를 고려하여 실행시간에 iteration들을 할당하는 동적 스케쥴링으로 구분된다. 정적 스케쥴링은 각 iteration들의 분산이 불규칙적인 경우 load imbalance가 발생되며, 동적 스케쥴링은 동기화 오버헤드 및 병목현상을 수반한다. 따라서, 스케쥴링에서는 서로 독립된 각 iteration들의 실행시간에 따른 최소의 동기화 오버헤드와 load balnce를 달성하는데 초점을 두며, 그에 따른 일반적인 기존 스케쥴링 방법들을 살펴보면 다음과 같다.

Self-Scheduling(SS)[4,6]는 모든 iteration들이 중앙 작업 큐에 놓여지고, 프로세서가 idle하면, 중앙 작업 큐로부터 동기화 방법으로 루프 인덱스를 증가하며 하나의 iteration을 선택하는 가장 단순한 스케쥴링 방법이다. 따라서, 프로세서들이 각각 한 iteration내에 끝나기 때문에 load balance에 관해서는 최적의 동적 스케쥴링 방법이다. 그러나 병렬 iteration 수인 N dispatch operation이 발생하며, 공유 메모리의 상호 배타적인 접근이 빈번하여, 오버헤드와 병목현상이 발생된다.

Chunk Self-Scheduling(CSS)[1]는 메모리의 상호 배타적인 접근을 줄이기 위해서 idle 프로세서에게 iteration의 그룹인 고정된 크기 즉, k의 chunk를 할당하므로서 동기화 오버헤드를 줄이려고 하였다. 그러나 실행시간에 조차 알려지지 않은 각 iteration의 분포 때문에 특히, 불규칙 루프에서 load imbalance가 발생되며, 또한 최적의 chunk를 구하기가 어렵다.

Guided Self-Scheduling(GSS)[2]는 중앙 작업 큐에 있는 실행될 iteration을 idle 프로세서에게 할당할 때, 처음에는 큰 chunk를 할당하므로서 스케쥴링 오버헤드를 감소시키고, 실행시간 동안 chunk의 크기를 계속 감소하여 load balance를 달성하려고 하였다. GSS에서 chunk의 크기는 다음과 같이 결정된다. $I_c = 1,2,3,4,\ldots.N$ 에 대해

$$R_0 = N, R_{i+1} = R_i - x_i$$

$$x_i = \lceil \frac{R_i}{P} \rceil \qquad (1)$$

N : 총 iteration 수, P : 프로세서 수

그러나 공유 메모리로부터 반복적으로 chunk를 계산하고 할당하기 위해서 상호 배타적인 빈번한 메모리의 접근으로 많은 동기화 오버헤드가 발생된다. 또한 초기에 너무 큰 chunk가 프로세서에게 할당되기 때문에 iteration들의 실행시간이 선형적으로 감소하거나, 처음 몇몇 iteration들의 실행시간이 큰 경우, 최적의 실행 시간내에 끝나지 않는다.

Factoring[5]은 iteration들의 chunk가 P개의 프로세서로 스케쥴링된 후, 중앙 작업 큐에 충분한 iteration들의 수 즉, PF_0를 남겨 load balance를 달성하는 확률을 높이려고 하였다. Factoring에서는 iteration들에 대해서 P개의 동일한 크기 chunk들의 batch들로 스케쥴 된다. 따라서, 각 batch는 중앙 작업 큐에 남이있는 iteration들의 고정된 비율이 된다. Factoring에서 각 chunk는 다음과 같이 결정된다. 서로 독립된 iteration들 $I_c = 1,2,3,4,\ldots.N$ 에 대해

$$R_0 = N, R_{j+1} = R_j - PF_j$$

$$F_j = \lceil \frac{R_j}{x_j P} \rceil , \quad x_j = 2 \qquad (2)$$

Factoring은 초기 몇몇 iteration들의 분산이 클 때, load imbalance 때문에 최적의 종료 시간을 달성하지 못한다.

Affinity Scheduling[3]은 초기에 iteration에 대해서 각 프로세서에게 정적으로 $\lceil \frac{N}{P} \rceil$인 chunk를 할당하고, 각 프로세서들이 독립적으로 스케쥴링을 수행하므로서 동기화에 대한 요구를 최소화 하려고 하였다. 만일 프로세서가 idle하면, 가장 많이 남은 프로세서의 지역 메모리를 찾아 $\lceil \frac{1}{P} \rceil$의 iteration을 옮겨 실행하므로서 load balance를 달성하려고 하였다. 그러나 초기 chunk를 각 프로세서에게 할당할 때, 자료 국부성을 고려 하지 않았다. 따라서, 각 chunk내 iteration들의 분포가 상이하여 다른 프로세서 메모리의 접근 횟수가 많아지고, 접근된 메모리로부터 적은 iteration들만을 취하므로서 많은 오버헤드가 발생된다.

3. 개선된 병렬 스케쥴링 알고리즘

본 논문에서는 자료 국부성과 프로세서 동족성을 고려하여 load balace를 달성하고, 최소의 메모리 경쟁을 통해 스케쥴링 오버헤드를 최소화하여 최적의 스케쥴링을 달성할 수 있는 새로운 알고리즘을 제안한다. 기존의 대부분 스케쥴링 알고리즘들은 서로 독립된 N개의 iteration들인 $[1,2,3,\ldots,N-2,N-1,N]$에 대해서 서로 독립된 임의 변수로 취급하여 전역 메모리로부터 프로세서가 idle할 때마다 chunk 크기를 반복적으로 계산하여 프로세서에게 할당한다. 이는 빈번한 메모리의 상호 배타적인 접근과 반복된 chunk의 계산 및 할당으로 chunk의 수에 비례하는 스케쥴링 오버헤드를 유발한다. Affinity Scheduling[3]은 자료 국부성을 충분히 고려하지 않고 $\frac{N}{P}$의 chunk를 정적으로 할당 하므로서 빈번한 메모리의 상호 배타적인 접근으로 많은 오버헤드를 발생되어 효율적인 스케쥴링을 달성할 수 없다. 따라서, 최적의 스케쥴링을 달성하기 위해서는 각 iteration들에 대해, 자료 국부성을 충분히 고려하여 분해하므로서 분해된 각 그룹간에 최소의 상이한 실행시간을

유지하도록 해야한다. 또한, 분해된 그룹들을 프로세서 동족성을 고려하여 프로세서들의 지역 메모리에 정적으로 할당하고, 각 프로세서는 자신의 그룹으로 부터 연속된 iteration들의 범위에 대해서 단일 루프 인덱스로 독립적인 실행을 하므로서 전체적으로 최소의 스케쥴링 오버헤드와 load balance를 달성할 수 있다.

즉, $[1,2,3,\ldots,N-3,N-2,N-1,N]$에 대해 자료 국부성을 고려하여 컴파일 시간에 정적으로 분해 작업을 수행한다. 분해 작업의 기본 개념은 주어진 iteration들을 제어키에 의해 실행시간 분포가 비슷한 여러개의 그룹으로 분해하고, 각 그룹들을 프로세서에 의해 독립적으로 스케쥴링을 수행하게 하기 위한 것이다. 이때, 제어 키는 프로세서의 수가 된다. 예를 들면, 프로세서가 4인 다중프로세서 시스템인 경우를 고려 해보자. 첫째, 제어 키는 프로세서의 수가 되기 때문에 4가 된다. 따라서, 첫 번째 iteration으로 부터 제어 키 만큼 떨어진 iteration들을 하나의 그룹으로 묶으면 [그림1]와 같이 네 개의 그룹으로 분해된다.

$[1,2,3,\ldots,N-3,N-2,N-1,N]$에 대해

$G_1 = [1, 1+CK, 1+2CK, 1+3CK, \ldots, N-3]$

$\Rightarrow X[1], X[5], X[9], X[13]\ldots$

$G_2 = [2, 2+CK, 2+2CK, 2+3CK, \ldots, N-2]$

$\Rightarrow X[2], X[6], X[10], X[14]\ldots$

$G_3 = [3, 3+CK, 3+2CK, 3+3CK, \ldots, N-2]$

$\Rightarrow X[3], X[7], X[11], X[15]\ldots$

$G_4 = [4, 4+CK, 4+2CK, 4+3CK, \ldots, N]$

$\Rightarrow X[4], X[8], X[12], X[16]\ldots$

CK : 제어 키(Control Key), X : 요소(Element)

[그림1] 제어키에 의해 분해된 iteration들의 그룹

따라서, G_i번째 그룹의 i번째 요소를 X라하면, 그 순서 위치는 다음과 같다.

$$X[(i-1)*CK+G_i] \qquad (6)$$

둘째, 분해된 그룹들은 각 프로세서에게 할당되고, 프로세서들은 각각 자신의 지역 메모리로 부터 독립적으로 iteration들에 대해 스케쥴링을 수행한다. 따라서, 프로세서들은 iteration들을 독립적으로 dispatch하기 때문에 병목현상이 발생되지 않는다. 셋째, 만일 프로세서 P_i가 idle하면, 프로세서 P_i는 나머지 P-1개의 프로세서를 분석하고, iteration들이 가장 많은 프로세서로의 iteration들에 대해서 다시 분해 작업을 수행한다. 이때, 분해될 iteration들은 원래 수행하고 있던 프로세서와 idle 프로세서 P_i 즉, 2개의 프로세서에 의해 실행되므로 제어 키는 2가 된다. 분해된 그룹은 각 프로세서에 의해 독립적으로 스케쥴링이 반복수행된다. 예를들면, 프로세서 P_1이 idle하고, P_4가 가장 많은 iteration들을 가지고 있다고 가정하면, 프로세서 P_4에 남아있는 iteration들에 대한 분해 과정은 [그림2]와 같다. 즉, P_4에 남아있는 iteration들

$[24,28,32,36,\ldots,N-4,N]$에 대해

$G_1 = [1, 1+CK, 1+2CK, 1+3CK, \ldots, N-4]$

$\Rightarrow X[1], X[3], X[5], X[7]\ldots$

$G_2 = [2, 2+CK, 2+2CK, 2+3CK, \ldots, N]$

$\Rightarrow X[2], X[4], X[6], X[8]\ldots$

[그림2] 가장 많이 남은 iteration들의 재분해

분해된 각 그룹에 대해서 프로세서 P_1은 G_1을 P_4은 G_2를 독립적으로 스케쥴링을 수행한다. 그 과정은 [그림3]과 같다.

```
1. loop_initialization(N, CK, P)
     CK ← P //실행 처리할 프로세서 수를 제어키에 대입//
2. for I ← 1 to CK
3.    for J←1 to N  by P //iteration들을 P개의 그룹으로 분해//
          G_i ← X(j)// G_i ={ G_1, G_2, ..., G_p}
      end
      P ← P-1
    end
4. for I ← 1 to CK //지역 메모리로 그룹 iteration들을 할당//
     P_i←G_i
   end
5. loop
     for I ← 1 to N //각 프로세서의 독립적인 스케쥴링//
       range ← X(i)
       if (range = empty)
         mlq= find_most_loaded_local_queue
         if (mlq=null) break;
         CK ← 2 //제어키에 2를 대입//
            for I ← 1 to CK-1
              for J ← 1 to N  by 2 //재분해 작업//
                 G_i ← X(j)
              end
            end
          P_i←G_i
     end
   forever
```

[그림3] 개선된 병렬 루프 스케쥴링 알고리즘

4. 실험 평가

본 논문에서 제안된 알고리즘의 성능평가를 위해서 2개의 벤치마크 프로그램을 선택하였다. 성능 평가는 128개의 DEC Alpha chip으로 구성된 CRAY T3E-900 CL128a-128인 MPP 머신에서 수행된다. 선택된 프로그램들은 2, 4, 8, 16개의 프로세서에서 총 실행시간에 대해

평가하며, 기존 스케쥴링 알고리즘들 중에서 성능이 우수한 것으로 평가된 GSS, Factoring, Affinity와 비교 평가한다. 벤치마크 프로그램은 다음과 같다.

```
* DOALL 10 I=1, N*N
      DOSERIAL 20 K=I, N*N
        A(I)=A(I)+X*B(K)*C(K-I)
    20 ENDSERIAL
10 ENDOALL
```

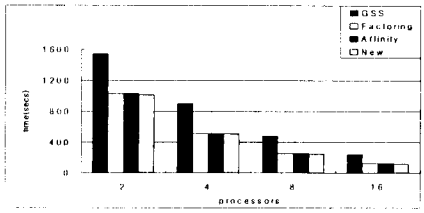
[그림4] Adjoint-convolution Program

위 프로그램들은 N*N개의 병렬 iteration들이 $O(n^2-i)$에 비례하는 실행시간을 갖는다.

```
* DOALL 10 I=1, N
    IF [ C then {350} ]
    DOSERIAL 20 K=1, M
      {10}
    20 ENDSERIAL
  10 ENDOALL
```

[그림5] 불규칙 루프

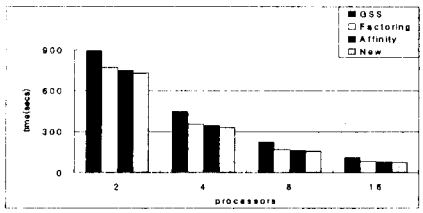
위 프로그램은 N개의 병렬 iteration들이 C의 조건에 따라 350개의 직선 코드를 실행한 후, M번의 순차코드를 실행한다.



[그림6] Adjoint-convolution 성능평가

<표1> Adjoint-convolution 실행시간

프로세서수	GSS		Factoring		Affinity		새로운방법	
	실행시간	실행율	실행시간	실행율	실행시간	실행율	실행시간	실행율
2	1538.77	65.88	1026.27	98.74	1033.56	98.04	1013.34	100
4	897.48	56.78	513.14	99.32	516.81	98.61	509.66	100
8	480.88	52.05	256.67	97.52	258.48	96.83	250.31	100
16	248.47	48.35	128.31	93.64	129.48	92.80	120.16	100



[그림7] 불규칙 루프 I≤90 성능평가

<표2> 불규칙 루프 I≤90 실행시간

프로세서수	GSS		Factoring		Affinity		새로운방법	
	실행시간	실행율	실행시간	실행율	실행시간	실행율	실행시간	실행율
2	895.74	81.62	771.93	94.71	749.78	97.51	731.15	100
4	447.86	74.03	355.07	93.37	345.92	95.84	331.56	100
8	223.91	70.24	169.73	92.66	164.46	95.63	157.28	100
16	111.95	68.34	82.95	92.23	79.72	95.97	76.51	100

Adjoint-convolution은 GSS는 처음 할당된 chunk의 분포가 너무 커 load imbalance가 발생되어 가장 나쁜 수행 결과를 보였으며, 제안된 방법은 Factoring과 Affinity에 비해서 약 3%와 4%의 성능 향상을 보였다. 불규칙 루프는 GSS와 Factoring은 load imbalance가 발생되며, 따라서 제안된 방법은GSS와 Factoring에 비해 각각 약 26%와 7%의 성능 향상을 보였다. Affinity는 각 정적으로 할당된 iteration들의 분포가 상이하여 각 프로세서의 load balance를 유지하기 위해서 과도하게 메모리에 접근으로 과도한 오버헤드가 발생된다. 따라서, 제안된 방법은 약 4%의 성능 향상을 보였다. 실험 결과에서, 제안된 방법은 기존 알고리즘에 비해 최소의 오버헤드와 load balance를 유지할 수 있었으며, 많아야 1회의 메모리 접근으로 모든 iteration들의 수행을 완료되었다. 특히, 제안된 방법은 iteration들의 분포가 선형적으로 감소하거나 처음 몇몇 iteration들의 분포가 클 때, GSS나 Factoring보다 좋은 성능 향상을 보였으며, 불규칙적인 루프에서 Affinity보다 더 좋은 성능향상을 보였다.

5. 결론

최적의 스케쥴링을 달성하기 위해서는 스케쥴링 오버헤드를 최소화 하고, load balance를 달성해야 한다. 본 논문에서는 동적 및 정적 스케쥴링을 모두 고려한 최적의 스케쥴링을 위해, 서로 독립된 iteration들에 대한 자료 국부성을 고려하여 iteration들을 분해하고, 프로세서 동족성을 고려하여 정적으로 각 프로세서에게 할당하므로서 최소의 동기화 오버헤드및 load imbalance를 달성하는 스케쥴링 알고리즘을 제안 하였다. 또한 제안된 알고리즘은 실험 결과 전체적으로 GSS보다 약 35%, Factoring보다 약 6%, 그리고 Affinity보다 약 5%의 향상된 스케쥴링을 수행할 수 있었다. 향후 과제로는 본 논문에서 제안된 스케쥴링 알고리즘을 토대로 분산 메모리 시스템에서도 최적의 스케쥴링을 달성할 수 있는 알고리즘에 대한 연구가 필요하다.

참고 문헌

[1] C. Kruskal and A. Weiss, "Allocating independent subtasks on parallel processings," IEEE Transactions on Software Engineering, SE-11, October 1989.

[2] C. Polychronopoulos and D. Kuck, "Guided self-scheduling : A practical self-scheduling scheme for parallel supercomputers,"IEEE Trans. Comput. vol.C-36,pp. 1425-1439, December. 1987.

[3] E. P. Markatos, T. J. LeBlanc, "Using Processor Affinity in Loop Scheduling on Shared-Memory Multiprocessor," IEEE Transactions Parallel and Distributed Systems, vol. 5, no. 4 April 1994.

[4] P. Tang and P. C. Yew, "processor self-scheduling for multiple-nested parallel loops,"in Proc. 1986 Int. Conf. parallel processing, pp.528-535, August. 1986.

[5] S. F. Hummal, I. Banicescu, C. Wang, and J. Wein, "Load balancing and data locality via Factoring: an Experimental Study,"Proc. Third Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers, pp. 85-89, Boleslaw K. Szymanski and Balaram Sinharoy(Editor), Kluwer Academic Publishers, Boston, May, 1995.

[6] Z. Fang, P. C Yew, P. Tang, and C. Q. Zhu, "Dynamic processor self-scheduling for general parallel nested loops," in Proc. 1987 Int, Conf. parallel processing, pp.1-10, August. 1987.