블록-순환으로 분배된 배열의 지역 주소 생성

권 오 영[†] · 김 태 근[†] · 한 탁 돈[††] · 양 성 봉[†††] · 김 신 덕[†††]

요 약

대부분의 데이터 병렬 언어들은 배열을 분배하는 방법을 제공하고 있다. 이들 중 블록-순환(block-cyclic) 분배가 가장 일반적인 데이터 분배 방법이다. 블록-순환 형태로 분할된 배열 구간 A(l:h:s) 중 각 프로세서가 자신의 메모리 영역에서 접근하는 A의 지역주소를 컴파일러 또는 실시간 시스템들이 생성하는 방안에 대한 연구가 이루어지고 있다. 이 논문에서는 블록-순환 분배된 배열에 대한 두 가지 지역 주소 생성 방법을 제안한다. 하나는 가상-블록 (virtual-block)을 변형한 simple scan 방법이고, 다른 하나는 지역 메모리 접근에 대한 정보를 포함하는 ΔM테이블을 선형시간에 생성하는 알고리즘이다. ΔM테이블 생성과 각 프로세서가 10,000개의 지역 배열 원소를 접근하는데 소요된 시간을 측정하는 실험을 하였다. 실험결과 simple scan 방법은 성능이 좋지 못하였다. 하지만 ΔM 테이블을 구성하는 방안은 다른 방법들 보다 빠른 시간에 수행이 완료되었다.

Generating Local Addresses for Block-Cyclic Distributed Array

Oh-Young Kwon[†] · Tae-Geun Kim[†] · Tack-Don Han[††]
Sung-Bong Yang[†††] · Shin-Dug Kim[†††]

ABSTRACT

Most data parallel languages provide the block-cyclic distribution (cyclic(k)) that is one of the most general regular distributions. In order to generate local addresses for an array section A(l:h:s) with block-cyclic distribution, efficient compiling methods or run-time methods are required. In this paper, two local address generation methods for the block-cyclic distribution are presented. One is a simple scan method that is modified from the virtual-block scheme. The other is a linear-time ΔM table that contains the local memory access information construction method. This method is simpler than other algorithms for generating a ΔM table. Experimental results show that a simple scan method has poor performance but a linear-time ΔM table generation method is faster than other algorithms in ΔM table generation time and access time for 10,000 array elements.

1. 서 론

High Performance Fortran (HPF)과 같은 대부분의 데이터 병렬 언어들은 배열을 분배하는 방법을 제공하고 있다. 이들 중 블록-순환 분배가 가장 일반적인 데이터 분배 방법이다[1]. 본 논문에서 표현된 *cyclic(k)*는 배열이 하나의 블록이 k개의 연속된 배열 원소로 이루어져 있고, 이 블록이 각 프로세서에 round-robin 형태로 분배되었음을 의미한다. 편의상 본 논문에서는 k를 블록의 크기라고 부르고, 이는 연속적인 k개의 배열 원소를 의미한다. (그림 1)은 *cyclic(4)*로 분배된 배열 A(0:159)가 4개의 프로세서에 분배된 모습을 보

† 정 회 원 : 한국전자통신연구원(ETRI) 네트워크컴퓨팅연구부 분산컴퓨팅연구팀
†† 종신회원 : 연세대학교 컴퓨터과학과 교수
††† 정 회 원 : 연세대학교 컴퓨터과학과 교수
논문접수 : 1998년 4월 17일, 심사완료 : 1998년 9월 21일

여주고 있다. 그림에 각 배열 원소의 전역 주소를 표시하였으며 그에 해당하는 지역주소를 전역 주소의 오른쪽하단에 표시하였다. 그리고, 각 원소중 어둡게 표시된 부분은 배열 구간 A(0:155:5)가 각 프로세서에서 접근이 발생하는 부분을 표시한다.

블록-순환 형태로 분배된 배열 구간 A(l:h:s) 중 각 프로세서가 자신의 메모리 영역에서 접근하는 A의 지역주소를 컴파일러 또는 실시간 시스템들이 생성하는 방안에 대한 연구가 이루어지고 있다. 여기서, *l*은 배열의 참조가 시작되는 전역 하한 값이며, *h*는 배열의 참조가 끝나는 전역 상한 값이고, *s*는 배열 참조간의 간격을 의미한다. (그림 1)에서 각 프로세서가 접근하는 지역주소는 *k*개의 state를 갖고, (그림 2)와 같이 상태전이가 일어나는 Finite State Machine (FSM)으로 표현할 수 있다. 각 프로세서는 시작 상태만 다를 뿐 (그림 2)의 상태전이 규칙을 따른다. 즉, 프로세서 P0은 시작 상태가 0이고, 프로세서 P3은 시작 상태가 3이다.

Chatterjee, *et al.* [1]은 이러한 사실에 기초하여

현재상태	다음상태	주소차
0	3	11
1	0	3
2	1	3
3	2	3

(그림 2) 지역 주소의 상태 전이 규칙
(Fig. 2) State Transition Rule of Local Addresses

Finite State Machine을 블록-순환 배열의 지역주소를 이용하여 생성하였다. (그림 2)와 같이 상태 전이 규칙을 표현한 테이블에 두 상태간의 지역 주소차를 포함하고 있는 테이블을 ΔM 테이블이라 한다. 이들의 알고리즘은 ΔM 테이블을 구성하는데 $O(k \log k)$의 시간을 필요로 함을 보였다. 여기서, *k*는 블록 크기이다.

Kennedy, *et al.* [2]와 Thirumalai와 Ramanujam [3]은 배열 구간의 접근 형태를 2차원 공간내의 integer lattice로 간주하고, lattice를 구성하는 basis vector를 이용하여 각 프로세서의 지역 주소 접근 패턴 (access pattern)을 찾는 알고리즘을 제안하였다. 예를 들면, 프로세서 P0은 0, 3, 2, 1의 순서로 접근이 이루어지고 이들의 지역 주소는 각각 0, 11, 14, 17이 된다. 그리고,

P0				P1				P2				P3			
	1 1	2 2	3 3	4 0		6 2	7 3	8 0	9 1		11 3	12 0	13 1	14 2	
16 4	17 5	18 6	19 7		21 5	22 6	23 7	24 4		26 6	27 7	28 4	29 5		31 7
32 8	33 9	34 10		36 8	37 9	38 10	39 11		41 9	42 10	43 11	44 8		46 10	47 11
48 12	49 13		51 15	52 12	53 13	54 14		56 12	57 13	58 14	59 15		61 13	62 14	63 15
64 16		66 18	67 19	68 16	69 17		71 19	72 16	73 17	74 18		76 16	77 17	78 18	79 19
	81 21	82 22	83 23	84 20		86 22	87 23	88 20	89 21		91 23	92 20	93 21	94 22	
96 24	97 25	98 26	99 27		101 25	102 26	103 27	104 24		106 26	107 27	108 24	109 25		111 27
112 28	113 29	114 30		116 28	117 29	118 30	119 31		121 29	122 30	123 31	124 28		126 30	127 31
128 32	129 33		131 35	132 32	133 33	134 34		136 32	137 33	138 34	139 35		141 33	142 34	143 35
144 36		146 38	147 39	148 36	149 37		151 39	152 36	153 37	154 38		156 36	157 37	158 38	159 39

(그림 1) A(0:155:5)의 분배 형태
(Fig. 1) Distribution Form of A(0:155:5)

프로세서 P1은 1, 0, 3, 2의 순서로 접근이 이루어지고 이들의 지역 주소는 1, 4, 15, 18이 된다. Kennedy, *et al.*은 이러한 정보를 이용하여 각 프로세서의 ΔM 테이블을 구성하지만, Thirumalai과 Ramanujam는 ΔM 테이블 대신에 각 프로세서의 접근 패턴에 대한 정보를 저장한 패턴 테이블을 구성하는 것이 두 알고리즘의 차이점이다. 이들의 알고리즘의 수행에는 $O(k)$ 시간이 소요된다. 그러나, 지역 주소를 생성해낼 때 패턴 테이블의 경우는 약간의 연산이 더 필요하다[3].

Thirumalai와 Ramanujam는 [3]에서 제시된 basis vector를 찾는 알고리즘 대신에 블록 크기 k, 프로세서의 개수 p, stride s를 이용하여 basis vector를 closed form으로 유도하는 방안을 제안하였다[4]. Closed form으로 유도된 basis vector는 [2]와 [3]에서 basis vector를 구하는 부분에 대한 연산이 없어지므로 접근 패턴을 빨리 생성할 수 있다. 하지만, p, k, s의 관계에 따라 basis vector를 생성하는 closed form이 여러 개가 존재하는 단점이 있다.

Gupta, *et al.* [5]은 *cyclic(k)*를 가상 프로세서에 배열을 블록 크기가 k인 블록 분할을 수행하고, 가상 프로세서를 *cyclic*하게 프로세서에 매핑하는 *virtual-block* 방법과 가상 프로세서에 배열을 *cyclic* 분배하고, 가상 프로세서를 k개씩 프로세서에 블록 매핑하는 *virtual-cyclic* 방법으로 해석하였다. 이들이 제안한 방법을 이용하면 지역 주소 생성에 많은 시간을 요하게 된다. 이러한 사실은 [6]에 제시되어있다. 특히, $s>k$인 경우에도 모든 가상 프로세서를 참조하므로 상당히 많은 시간을 요하게 된다.

본 논문은 *virtual-block* 방법을 기반으로 하나의 프로세서에 포함되어 있는 모든 가상 프로세서를 scan하면서 지역 주소를 생성하는 단순 주소 생성 방법을 제안한다. 이 방법은 $s>k$인 경우 *virtual-block* 방법이 갖고 있는 단점을 그대로 가지고 있다. 대신 전체 가상 프로세서의 참조대신 자신의 프로세서에 포함된 가상 프로세서들만 참조하면 되는 장점이 있다.

일반적으로 $s>k$이면 지역 주소를 생성하는데 ΔM 테이블이 사용된다. 만일 x_0와 x 가 동일 프로세서 상에 연속된 지역 주소 접근이고 x_0의 지역 주소를 알고 있다고 가정하자. 이 정보를 이용하여 x에 대한 지역 주소를 $O(1)$ 시간 안에 찾을 수 있으면 ΔM 테이블은 $O(k)$ 시간 안에 구성할 수 있다. 본 논문에서는 x를 찾는 수식이 제시되었고 ΔM 테이블을 이러한 수식을 이용하여 선형 시간 안에 구성하는 방안을 제시하였다.

본 논문의 2절에서는 *virtual-block* 방법을 이용한 단순 주소 생성 방법을 설명하고, 3절에서는 스트라이드 s, 블록크기 k, 그리고 프로세서의 개수 p를 이용하여 ΔM 테이블을 선형시간에 구성하는 방법에 대하여 설명한다. 4절에서 기존의 방법들과 비교한 실험결과를 기술하였고, 5절에서 결론 및 향후연구방향을 논의한다.

2. 단순 주소 생성

$A(l:h:s) = X$는 (그림 3)에서 보여지는 것과 같이 일반화된 루프로 변화된다.

```
DO i = 0, ⌊(h−l)/s⌋
  A( l+s·i ) = X
END DO
```

(그림 3) A(l:h:s) = X를 위한 일반화된 루프
(Fig. 3) Normalized Loop for A(l:h:s) = X

$A(l:h:s)$는 배열 A(0:N)의 일부분이라고 가정하고, 즉 $0 \le l < h \le N$의 관계를 만족하고, A는 p개의 프로세서에 *cyclic(k)* 방법으로 분할되었다고 가정하자. A(0:N)의 배열을 크기 k 의 블록들로 분배하려면 $\lceil (N+1)/s \rceil$ 개의 가상 프로세서들이 요구된다. 가상 프로세서를 V_n, ($n=0,\ldots,\lfloor (N+1)/s \rfloor$) 이라 하자. 가상프로세서를 p개의 프로세서에 순환 배치함으로써 배열의 블록-순환 분할을 완료한다. 임의의 프로세서 m은 가상 프로세서 V_n들을 포함하는데 이때 V_n은 $w \cdot p+m$, $w=0, \ldots, \lfloor h/(p \cdot k) \rfloor$를 만족하는 가상 프로세서들이다. w는 프로세서 m안의 V_n의 위치를 나타낸다.

(그림 3)의 루프에서 임의의 i가 생성하는 전역 주소 G_a ($=l+i \cdot s$)의 지역 주소 L_a는 가상 프로세서들을 사용함으로써 찾을 수 있다. G_a는 가상 프로세서 $V_n = \lfloor G_a/k \rfloor$에 포함된다. 만약 V_n을 $w \cdot p+m$으로 표현되면, 가상 프로세서 V_n에서 G_a에 해당하는 지역 주소는 $G_a - V_n \cdot k$이 된다. V_n를 포함하고 있는 프로세서 m은 이미 $w \cdot k$개의 배열 원소를 포함하고 있다. 그러므로, 프로세서 m이 가지고 있는

A[G_a]배열 원소에 대한 지역 주소 L_a는 $w \cdot k + G_a - V_n \cdot k$이다. 예를 들어 (그림 1)에서 전역 주소 70은 가상 프로세서 V_{17}에 포함되어 있다. V_{17}은 프로세서 1에 속하게 된다. 그러므로, 전역 주소 70은 프로세서 1에서 $4 \cdot 4 + 70 - 17 \cdot 4 = 18$의 지역주소에 해당된다.

전역 하한 l을 포함하는 가상프로세서를 $V_l = w_{\min} \cdot p + p_l$이라 하자. 여기서 p_l은 A[l]의 원소를 갖는 프로세서를 의미한다. 전역 상한 h는 가상 프로세서 $V_h = w_{\max} \cdot p + p_h$ 포함된다고 하자. 여기서 p_h는 A[h]의 원소를 갖는 프로세서를 나타낸다. 각 프로세서 m은 가상 프로세서 V_l과 V_h가 제공하는 정보를 이용하여 자신의 $w_{\min}$과 $w_{\max}$을 결정할 수 있다. 프로세서 m은 $w_{\min} \cdot p + m$에서 $w_{\max} \cdot p + m$까지의 가상 프로세서들을 포함한다. 가상 프로세서 V_n이 프로세서 m에 포함되고, 임의의 i가 $\lceil (V_n \cdot k - l)/s \rceil \le i < \lceil ((V_n + 1) \cdot k - l)/s \rceil$의 조건을 만족시킨다면 전역 주소 $l + s \cdot i$의 지역 주소는 프로세서 m상에서 $w \cdot k + l + s \cdot i - V_n \cdot k$이다. 이러한 정보를 이용하여 (그림 3)의 일반화된 루프는 쉽게 SPMD 코드로 변환할 수 있다. (그림 4)는 변환된 SPMD 코드이다.

```
1: m = get_mypid();

2: v_l = ⌊l/k⌋; w_min = v_l div p; p_l = v_l mod p;
3: if (m < p_l) w_min = w_min + 1;

4: v_h = ⌊h/k⌋; w_max = v_h div p; p_h = v_h mod p
5: if (m ≥ p_h) w_max = w_max − 1;

6: if (m == p_l) {
7:    t = (v_l − w_min) · k − l;
8:    iend = ⌈(min((v_l + 1) · k, h + 1) − l/s⌉;
9:    for (i = 0; i < iend; i++)
10:      A[s · i − t] = X;
11:   w_min = w_min + 1
12: }

13: v_n = w_min · p + m
14: for (w = w_min; w ≤ w_max; w++) {
15:   t = (v_n − w) · k − l;
16:   iend = ⌈((v_n + 1) · k − l)/s⌉;
17:   for (i = ⌈(v_n · k − l)/s⌉; i < iend; i++)
18:     A[s · i − t] = X;
19:   v_n = v_n + p;
20: }

21: if ((m == p_h) and (v_l ≠ v_h)) {
22:   t = (v_h − (w_max + 1)) · k − l;
23:   iend = ⌈(h + 1 − l)/s⌉;
24:   for (i = ⌈(v_h · k − l)/s⌉; i < iend; i++)
25:     A[s · i − t] = X;
26: }
```

(그림 4) A(l:h:s) = X의 SPMD 코드
(Fig. 4) SPMD code for A(l:h:s) = X

3. 선형시간 ΔM 테이블 생성 방법

블록순환 방식으로 분할된 배열에서 $s > k$이면 참조가 발생하지 않는 블록이 존재한다. (그림 1)의 P0에서 A[16]에서 A[19]에 해당되는 두 번째 블록은 배열 참조가 일어나지 않는다. 이러한 경우는 연속적인 지역 메모리 접근에 대한 정보를 포함하는 ΔM 테이블이 사용된다. 이 절에서는 선형시간에 ΔM 테이블을 구성하는 알고리즘을 기술한다.

3.1 $k < s \le p \cdot k$일 때 ΔM 테이블 생성

$\delta = s \bmod (p \cdot k)$이고, $r = (p \cdot k) \bmod \delta$라 하자. 프로세서들을 x축으로, 배열이 차지하고 있는 메모리 부분을 y축으로 나타낸다면 배열 섹션을 위한 요소들은 $x + r \cdot y = c$ 직선의 family상에 존재하게 된다. (그림 1)의 경우 배열 구간의 원소들이 $x + y = c$에 존재한다. 이러한 사실에 기반하여 주어진 점 $(x_0, 0)$에 대해서 그 뒤의 정수점 (x, y)는 (1) 이나 (2) 직선 상에 존재함을 알 수 있다.

$$x + r \cdot y = x_0 \tag{1}$$

$$x + r \cdot y = x_0 + \delta \tag{2}$$

직선의 기울기가 음수이기 때문에 (x, y)가 (1)상에 존재한다면 x의 범위는 $0 < x \le x_0$이다. 만약 (x, y)가 (2)상에 존재한다면 x의 범위는 $0 < x \le k$이다. x의 범위에 따라서 (1)의 y 범위는 (3)이 되고 (2)의 y 범위는 (4)가 된다.

$$0 < y \left(= \frac{x_0 - x}{r}\right) \le \frac{x_0}{r} \tag{3}$$

$$\frac{x_0 + \delta - k}{r} < y \left(= \frac{x_0 + \delta - x}{r}\right) \le \frac{x_0 + \delta}{r} \tag{4}$$

만약 $\lfloor x_0/r \rfloor \ge 1$이면, (x, y)는 (1) 위에 존재하고, 그렇지 않으면 (x, y)는 (2) 위에 존재한다. (1)에서 y는 1이 되고 x는 $x_0 - r$이 된다. (2)에서 y는

$\lfloor (x_0+\delta-k)/r \rfloor +1$이고, x는 $x_0+\delta-r\cdot y$가 된다. 만약 $r=0$이면, x와 x_0의 위치는 x 축 상에서 같고 $k<s\leq(p\cdot k)$이므로 y는 1이 된다. offset 즉, (x, y)와 $(x_0, 0)$사이의 지역 주소들간의 차는 다음과 같이 계산된다.

$$offset = y\cdot k-(x_0-x) \tag{5}$$

$x_0\in\{0,\ldots,k-1\}$인 모든 x_0에 대하여, (1)부터 (4)까지의 수식들을 사용하여 $(x_0, 0)$와 (x, y)의 쌍을 구하여 ΔM 테이블을 구성할 수 있다. (그림 5)는 stride가 $k<s\leq p\cdot k$일 때 선형시간에 ΔM 테이블을 구성하는 알고리즘을 보여주고 있다.

```
1:  δ = s mod (p · k);  r = (p · k) mod s;
2:  if (r==0)
3:    for ( x0=0;  x0<k;  x0++) {
4:      y = 1;  ΔM[ x0].next = x0;  ΔM[ x0].offset = k;
5:    }
6:  else {
7:    for ( x0=0;  x0<k;  x0++) {
8:      if ( ⌊ x0/r ⌋ ≥1) {
9:        y = 1;  ΔM[ x0].next = x0-r;
10:     }
11:     else {
12:       y = ⌊ (x0+δ-k)/r ⌋ ;
13:     ΔM[ x0].next = x0+δ-r·y ;
14:     }
15:     ΔM[ x0].offset = y·k - ( x0- ΔM[ x0].next);
16:   }
17: }
```

(그림 5) $k<s\leq p\cdot k$일 때 ΔM 테이블 구성 알고리즘
(Fig. 5) ΔM *Table Constructing Algorithm if* $k<s\leq p\cdot k$

3.2 $s>(p\cdot k)$ 일 때 ΔM 테이블 생성

3.1절에서 $\delta=s \bmod (p\cdot k)$로 정의되었다. s는 임의의 정수인 q에 대해 $s=q\cdot p\cdot k+\delta$를 만족한다. 결국, $(s\cdot i) \bmod k=((q\cdot p\cdot k+\delta)i) \bmod k=(\delta\cdot i) \bmod k$이다. 이 결과는 stride s와 stride δ를 갖는 배열 원소의 위치가 x축 상에 같은 위치에 있다는 것을 나타낸다. 즉, $x+r\cdot y=c$인 직선의 집합은 $\rho\cdot x+r\cdot y=\phi$인 직선의 집합으로 변한다. (2)에서 $x_0=0$라 가정하면 (2)는 $x+r\cdot y=\delta$로 다시 쓸 수 있다. 이 직선은 $(\delta,0)$와 $(\delta-r,1)$인 두 점을 포함한다. 이 두 점은 각각 δ와 $((p\cdot k-r)/\delta+1)\delta$인 배열 원소의 전역 주소에 해당한다. 그러므로, $s>p\cdot k$인 경우 전역 주소 s와 $((p\cdot k-r)/\delta+1)s$는 임의의 직선 $\rho\cdot x+r\cdot y=\phi$ 상의 두 점 $(\delta,\ s \text{ div } (p\cdot k))$와 $(\delta-r,\ (s((p\cdot k+\delta-r)/\delta)) \text{ div } (p\cdot k))$에 해당된다. 여기서 σ를 $s \text{ div } (p\cdot k)$라하고, ω를 $s((p\cdot k+\delta-r)/\delta))$ div $(p\cdot k)$이라 하면 $\rho=\omega-\sigma$가 된다. (그림 1)에서 stride가 21일 되면, (그림 6)과 같이 배열 원소의 x축 상의 위치는 동일하지만 y축의 위치는 변화를 가져온다.

$s>p\cdot k$인 경우 주어진 점 $(x_0, 0)$에 대해서, 다음 점 (x, y)는 (6) 또는 (7)위에 존재한다.

$$\rho\cdot x+r\cdot y=\rho\cdot x_0 \tag{6}$$

$$\rho\cdot x+r\cdot y=r\cdot\sigma+\rho(x_0+\delta) \tag{7}$$

만약 $\lfloor (\rho\cdot x_0)/r \rfloor\geq\rho$이면 (x, y)는 (6) 위에 존재한다. 이 경우 y는 ρ이 되고 x 는 x_0-r이 된다. 만약 (x, y)가 (7) 위에 존재한다면 x는 $\rho\cdot x$로 커지기 때문에 (4)와 같이 단순한 범위 테스트를 사용해서는 정확한 y를 구할 수 없다. 이러한 경우는 수식 $\rho(k-1-j)+r\cdot y=r\cdot\sigma+\rho(x_0+\delta)$를 만족하는 최소의 음수가 아닌 j가 구해질 수 있다면 x의 위치는 $k-1-j$이다. 위의 방정식은 다음의 (8)과 같이 표현된다.

$$\rho\cdot j-r\cdot y=\rho(k-1)-r\cdot\sigma-\rho(x_0+\delta) \tag{8}$$

$\lambda=\rho(k-1)-r\cdot\sigma-\rho(x_0+\delta)$이고, $g=\gcd(\rho, r)$이라 하자. 그러면 g는 항상 λ로 나눌 수 있다. 결국, 음수가 아닌 최소 j는 $(1/g)\{\lambda\cdot\alpha+r\lceil(-\lambda\cdot\alpha)/r\rceil\}$이다. 여기서 $g=\rho\cdot\alpha+r\cdot\beta$이다[1]. y는 $k-1-j$를 (7)에 적용하여 결정한다.

만약 $r=0$이고, $\delta\geq k$이면, x_0와 x의 위치는 같은 x축 상에 존재한다. 배열 구간 $A(l:h:s)$의 접근 형태가 $\text{LCM}(s, p\cdot k)$의 주기로 반복되기 때문에[1], y는 $\text{LCM}(s, p\cdot k)/(p\cdot k)$가 된다. $r=0$이고 $1\leq\delta<k$이면 x의 위치는 x축 상에서 $(x_0+\delta) \bmod k$가 된다. 이때 y는 $(x_0+\delta)<k$이면 $y=\sigma$가 되고 그렇지 않으면 $y=\text{LCM}(s, p\cdot k)/(p\cdot k)-(\sigma\cdot x_0)/\delta$가 된다. (그림 7)은 $s>p\cdot k$일 때 ΔM 테이블 구성 알고리즘이다. (그림 5)와 (그림 7)의 알고리즘은 $O(k)$ 시간에 ΔM 테이블을 구성한다.

P0				P1				P2				P3			
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3
16	17	18	19	20		22	23	24	25	26	27	28	29	30	31
4	5	6	7	4		6	7	4	5	6	7	4	5	6	7
32	33	34	35	36	37	38	39	40	41		43	44	45	46	47
8	9	10	11	8	9	10	11	8	9		11	8	9	10	11
48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	
12	13	14	15	12	13	14	15	12	13	14	15	12	13	14	
64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
16	17	18	19	16	17	18	19	16	17	18	19	16	17	18	19
80	81	82	83		85	86	87	88	89	90	91	92	93	94	95
20	21	22	23		21	22	23	20	21	22	23	20	21	22	23
96	97	98	99	100	101	102	103	104		106	107	108	109	110	111
24	25	26	27	24	25	26	27	24		26	27	24	25	26	27
112	113	114	115	116	117	118	119	120	121	122	123	124	125		127
28	29	30	31	28	29	30	31	28	29	30	31	28	29		31
128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143
32	33	34	35	32	33	34	35	32	33	34	35	32	33	34	35
144	145	146		148	149	150	151	152	153	154	155	156	157	158	159
36	37	38		36	37	38	39	36	37	38	39	36	37	38	39

(그림 6) A(0:147:21)의 분할 형태
(Fig. 6) Distribution Form of A(0:147:21)

4. 실험 결과

2절과 3절에서 제안한 방법을 Sun Sparc Station 20을 이용하여 실험하였다. 실험에 사용된 프로그램은 **gcc** 컴파일러를 이용하여 실행 프로그램을 만들었다. 컴파일시 **-O** 최적화 옵션을 사용하였으며 수행시간을 측정하기 위하여 **gettimeofday()** 함수가 이용되었다. 32개의 프로세서를 가진 병렬 컴퓨터를 시뮬레이션하기 위하여 프로세서 id 값인 m을 0부터 31까지 변화시켜 가면서 각 프로세서들에 대해 ΔM 테이블의 생성시간과 ΔM 테이블을 이용하여 10,000개의 지역 배열 원소를 참조하는 코드의 수행시간을 측정하였다. 측정한 수행 시간들 중에 가장 긴 수행 시간을 갖는 프로세서의 수행 시간을 실험결과로 사용하였다. 다른 알고리즘과 비교를 위해 [2]와 [4]에 제시된 알고리즘을 구현하였고, 실험에 사용한 stride (s), 프로세서의 수 (p), 블록 크기(k)와의 관계도 [2]와 [4]에서 제시된 것을 사용하였다. (그림 8)는 구현된 알고리즘에 의해 측정된 ΔM 테이블 구성시간을 보여준다. 여기서 CF는 [4]에서 제안된 알고리즘을 나타내며 Rice는 [2]의 알고리즘을 의미한다. ΔM 테이블을 구성하기 위한 우리의 방법이 다른 알고리즘과 비교하였을 때 비슷하거나 빠른 것을 알 수 있다. Rice의 알고리즘의 경우 basis vector를 구성하는 부분이 추가되어 다른 두 방법에 비하여 블록의 크기가 커질수록 ΔM 테이블 구성에 많은 시간이 소요되고 있다. CF 방법은 블록 크기, 프로세서의 수, 그리고 stride 크기의 관계에 따라서 basis vector를 형성하는 closed form을 선택하는 과정이 필요하다. (그림 8) (b)는 CF 방법은 basis vector 선택 과정의 처리 시간이 변화가 많음을 보여준다. 즉, s와 k간의 관계가 고정된 (c) ~ (f)와 달리 계속 변함으로 각 블록에 적합한 basis vector를 유도하는데 많은 시간이 소요된다.

(그림 9)는 각 프로세서가 ΔM 테이블을 구성하고, 이를 이용하여 10,000개의 지역 배열 원소들을 참조하는데 소요된 시간을 보여준다. (그림 9)의 Simple은 2절에서 제안된 방법을 의미하고 Run은 [6]에서 제안된 최적화된 실시간 주소 생성 알고리즘을 나타낸다. 이 두 가지 방법의 수행시간은 다른 방법에 비하여 상당히 길어서 그림의 표현 범위를 벗어났다. 다만 (그림 9) (a)에서 Simple 방법만이 블록의 크기가 커질수록 수행시간의 개선이 있었으며, 블록 크기가 32이상인 경우에는 그림에서 수행 시간을 확인할 수 있었다. 즉, Simple 방법은 스트라이드 s의 값이 적고 블록의 크기

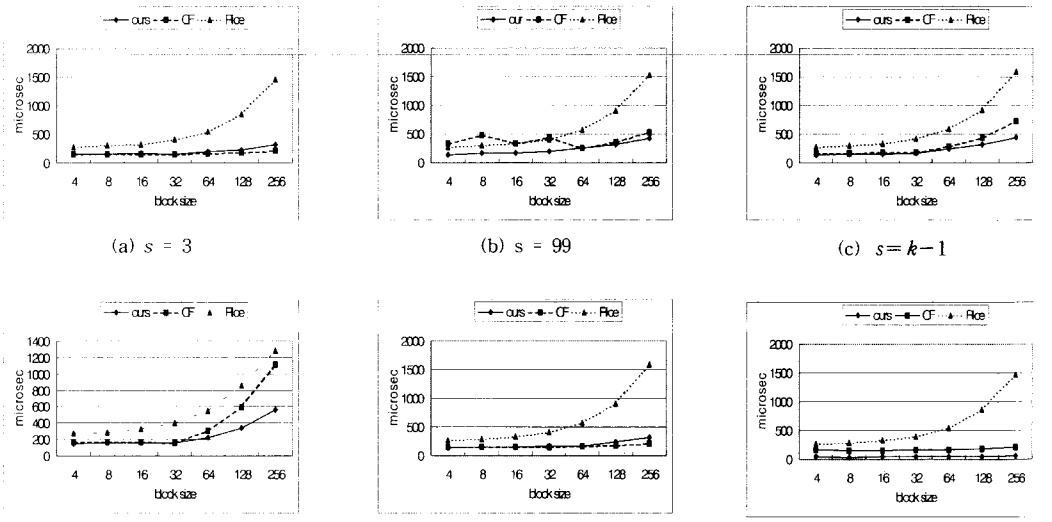
```
1:  δ=s mod (p·k);  r=(p·k) mod s;
2:  σ=s div (p·k);  ω=(s((p·k+δ−r)/δ)) div (p·k);
3:  ρ=ω−σ;;
4:  period = LCM(s,p·k)/(p·k);

5:  if (r==0) {
6:   if ( δ≥k )
7:    for ( x0=0; x0<k; x0++) {
8:     y = period; ΔM[ x0].next = x0; ΔM[ x0].offset = y·k;
9:    }
10:  else {
11:   for ( x0=0; x0<k; x0++) {
12:    x=x0+δ;
13:    if (x < k) y=σ;
14:    else {
15:     y = period − (σ·x0)/δ; x = x − r;
10:    }
11:    ΔM[ x0].next = x0+δ−r·y;
14:    ΔM[ x0].offset = y·k − ( x0− ΔM[ x0].next);
16:   }
17:  }
18: }
19: else {
20:  (g, α, β) ← Extended-Euclid( ρ, r);
21:  for ( x0=0; x0<k; x0++) {
22:   if ( ⌊(ρ·x0)/r⌋ < ρ ) {
23:    y=ρ;  ΔM[ x0 ].next = x0−r;
24:   }
25:   else {
26:    λ=ρ(k−1)−r·σ−ρ(x0+δ);
27:    j=(1/g){λ·α+r⌈(−λ·α)/r⌉};
28:    y=(1/r){r·σ+ρ(x0+δ)−ρ(k−1−j)};
29:    ΔM[ x0].next = k−1−j;
30:   }
31:   ΔM[ x0].offset = y·k − ( x0− ΔM[ x0].next);
32:  }
33: }
```

(그림 7) $s>p\cdot k$ 일 때 ΔM 테이블 구성 알고리즘
(Fig. 7) ΔM *Table Constructing Algorithm if* $s>p\cdot k$

인 k가 클수록 효과적인 방법임을 알 수 있다. 실험결과는 대부분의 경우에서 10,000개의 배열 요소들을 접근하는데 다른 알고리즘들 보다 3절에서 제안된 알고리즘의 수행시간이 빠르다는 것을 알 수 있다. 그림에서 알 수 있듯이 10,000개의 배열 원소에 대한 접근을 위하여 소요되는 시간은 ΔM 테이블을 구성하는 시간이 주도하고 있다. CF 방법은 현 상태와 이 상태의 주소를 갖는 패턴 테이블을 이용하므로, 지역 주소 계산 시 패턴 수를 곱하는 연산이 필요하다. 이러한 이유로, CF와 비슷한 ΔM 테이블 구성 시간을 갖는 제안된 방법이 실제 주소 계산에서는 대부분 좋은 성능을 보여주고 있다.

5. 결 론

본 논문에서 블록-순환방식으로 분할된 배열 구간의 효과적인 지역 주소 접근을 위하여 $O(k)$ 시간에 ΔM 테이블을 구성하는 새로운 알고리즘을 제안하였다. 기존에 발표된 동일한 시간 복잡도를 갖는 알고리즘과 실행시간을 비교하였을 때 제안한 방법이 다른 알고리즘 보다 우수하다. ΔM 테이블을 구성하는 경우 CF과 비슷한 성능을 보였다. 이 이유는 CF 방법은 ΔM 테이블에 필요한 정보를 전부 생성하지 않기 때문이다. 제안한 방법에 의하여 ΔM 테이블을 생성하고 이 테이블을 이용하여 10,000개의 배열 원소를 접근하는데 있어

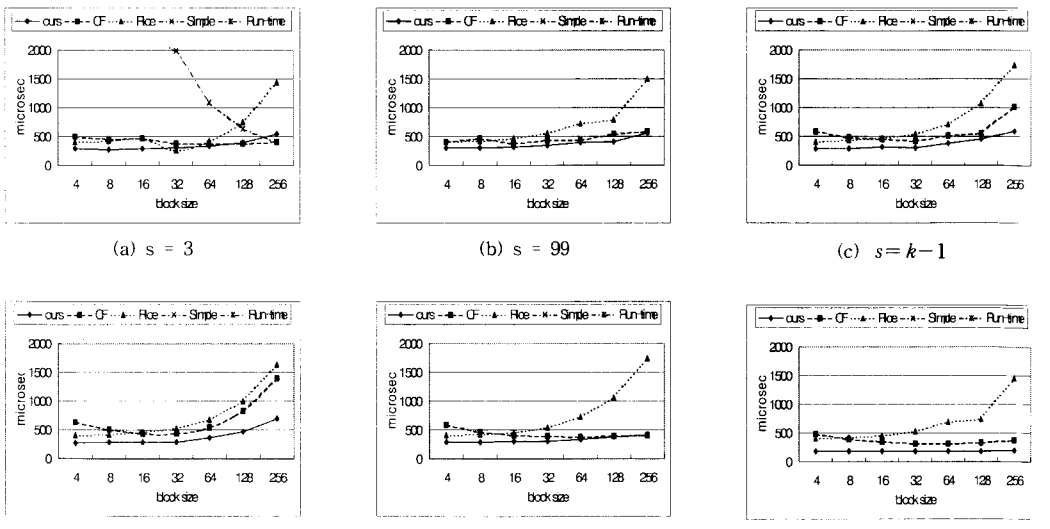


(a) s = 3 (b) s = 99 (c) $s=k-1$

(d) $s=k+1$ (e) $s=p\cdot k-1$ (f) $s=p\cdot k+1$

(그림 8) ΔM 테이블 생성 (μsec.)

(Fig. 8) ΔM table Generation (μsec.)



(a) s = 3 (b) s = 99 (c) $s=k-1$

(d) $s=k+1$ (e) $s=p\cdot k-1$ (f) $s=p\cdot k+1$

(그림 9) 10,000 개의 배열 원소 접근 (μsec.)

(Fig. 9) 10,000 Array Elements Access (μsec.)

다른 알고리즘 보다 제안한 방법이 우수한 성능을 보였다. 향후 서로 다른 블록크기 k_1과 k_2로 분할된 배열 A와 B에 대하여 $A(l_1:h_1:s_1)=B(l_2:h_2:s_2)$와 같은 치환문을 병렬화하기 위하여 본 논문에서 제안한 방법을 이용하여 각 프로세서가 필요로 하는 지역주소와 통신 집합을 생성하는 알고리즘을 연구하여야 한다.

참 고 문 헌

[1] S. Chatterjee, J.R. Gilbert, F.J.E. Long, R. Schreiber, and S.-H. Teng, "Generating Local Addresses and Communication Sets for Data Parallel Programs," Journal of Parallel and Distributed Computing, 26(1), April 1995.

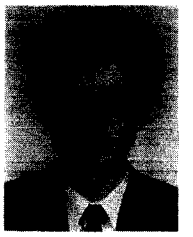
[2] K. Kennedy and N. Nedeljkovic, and A. Sethi, "A Linear-Time Algorithm for Computing the Memory Access Sequence in Data-Parallel Programs," Proc. of Fifth ACM SIGPLAN Symposium on PPoPP, Santa Barbara, CA, July 1995.

[3] A. Thirumalai and J. Ramanujam, "Fast Address Sequence Generation for Data-Parallel Programs Using Integer Lattices," Languages and Compilers for Parallel Computing, Lecture Notes in Computer Science, Springer-Verlag, 1996.

[4] A. Thirumalai and J. Ramanujam, "Efficient Computation of Address Sequences in Data Parallel Programs Using Closed Forms of Basis Vectors," Journal of Parallel and Distributed Computing, 38(2), November 1995.

[5] S.K.S. Gupta, S.D. Kaushik, C.-H. Huang, and P. Sadayappan, "On Compiling Array Expressions for Efficient Execution on Distributed-Memory Machines," Technical Report OSE-CISRC-4/94-TR19, Department of Computer and Information Science, The Ohio State University, April 1994.

[6] Ken Kennedy, Nenad Nedeljkovic, and Ajay Sethi, "Efficient Address Generation for Block-Cyclic Distributions," ICS '95, Barcelona, Spain, 1995.



권 오 영

oykwon@etri.re.kr

1990년 연세대학교 이과대학 전산과학과 졸업(이학사)

1992년 연세대학교 대학원 컴퓨터과학과 졸업(이학석사)

1997년 연세대학교 대학원 컴퓨터과학과 졸업(공학박사)

1997년~현재 ETRI 네트워크컴퓨팅연구부 분산컴퓨팅연구팀 선임연구원

관심분야 : 병렬화 컴파일러, 고성능 컴퓨터 구조, 병렬/분산 시스템

김 태 근

tkim@etri.re.kr

1987년 한양대학교 수학과 졸업 (학사)

1990년 뉴욕주립대학교 전산수학 졸업(석사)

1993년 뉴욕주립대학교 전산수학 졸업(박사)

1992년~현재 ETRI 네트워크컴퓨팅연구부 분산컴퓨팅연구팀 팀장

1997년~현재 ETRI 네트워크컴퓨팅연구부 자바센터 센터장

관심분야 : 병렬 컴파일러, 고성능 컴퓨터시스템, 병렬/분산 시스템

한 탁 돈

hantack@kurene.yonsei.ac.kr

1978년 연세대학교 공과대학 전자공학과(학사)

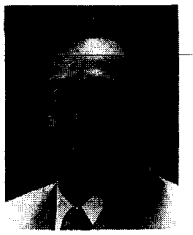
1983년 Wayne State University 컴퓨터공학(석사)

1987년 University of Massachusetts 컴퓨터공학(박사)

1987년~1989년 Cleveland 주립대학 조교수

1989년~현재 연세대학교 공과대학 컴퓨터과학과 교수

관심분야 : 병렬처리 컴퓨터 구조 및 알고리즘, 오류보정 시스템, VLSI 설계, HCI



양 성 봉

yang@kurene.yonsei.ac.kr

1981년 연세대학교 공과대학 요업공학과 졸업(학사)

1984년 University of Oklahoma 컴퓨터과학과(학부과정)

1986년 University of Oklahoma 컴퓨터과학과(석사)

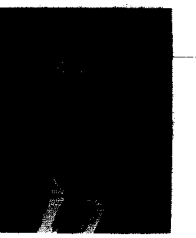
1992년 University of Oklahoma 컴퓨터과학과(박사)

1992년 8월~1992년 12월 University of Oklahoma Adjunct Assistant Professor

1993년 3월~1993년 8월 전주대학교 전자계산학과 전임강사

1994년 9월~현재 연세대학교 공과대학 컴퓨터과학과 조교수

관심분야 : 병렬 알고리즘, Genetic 알고리즘



김 신 덕

sdkim@kurene.yonsei.ac.kr

1982년 연세대학교 공과대학 전자공학과(학사)

1987년 University of Oklahoma 전기공학과(석사)

1991년 Purdue University 전기공학과(박사)

1993년 2월~1995년 2월 광운대학교 컴퓨터공학과 조교수

1995년 3월~현재 연세대학교 공과대학 컴퓨터과학과 부교수

관심분야 : 병렬처리 시스템, 컴퓨터 구조, Heterogeneous Computing