Code Generation for Complex Subscripts in Data-Parallel Programs

J. Ramanujam

Swaroop V. Dutta*

Arun Venkatachar

Louisiana State University

August 7, 1997

Web: http://www.ee.lsu.edu/jxr

Talk Outline

- Access to distributed arrays with complex subscript functions:
  - Multiple induction variables (MIV)
  - Coupled subscripts (CS)
- Efficient address generation for multiple induction variables
- Node code strategies for multiple induction variables
- Table based techniques for coupled subscripts
- RESET table for execution of coupled subscripts

Address Generation for Multiple Induction Variables

```
      REAL A(N)
!HPF$ PROCESSORS PROC(p)
!HPF$ ALIGN A(j) WITH T(j)
!HPF$ DISTRIBUTE (CYCLIC(k)) ONTO Proc::  T

DO i = 0, ..
   DO j = 0, ...
      A(i*s1 + l1 + j*s2 + l2) = ...
   ENDDO
ENDDO
```

Address Generation for multiple induction variables

Processor 0				Processor 1				Processor 2				Processor 3			
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127
128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143

$k = 4$, $p = 4$, $s_1 = 37$ and $s_2 = 2$

Processor 0	0	2	16	49	51	80	82	113	115	129

Processor 0	0	2	4	13	15	20	22	29	31	33

6

Computation of Start for $s \leq k$

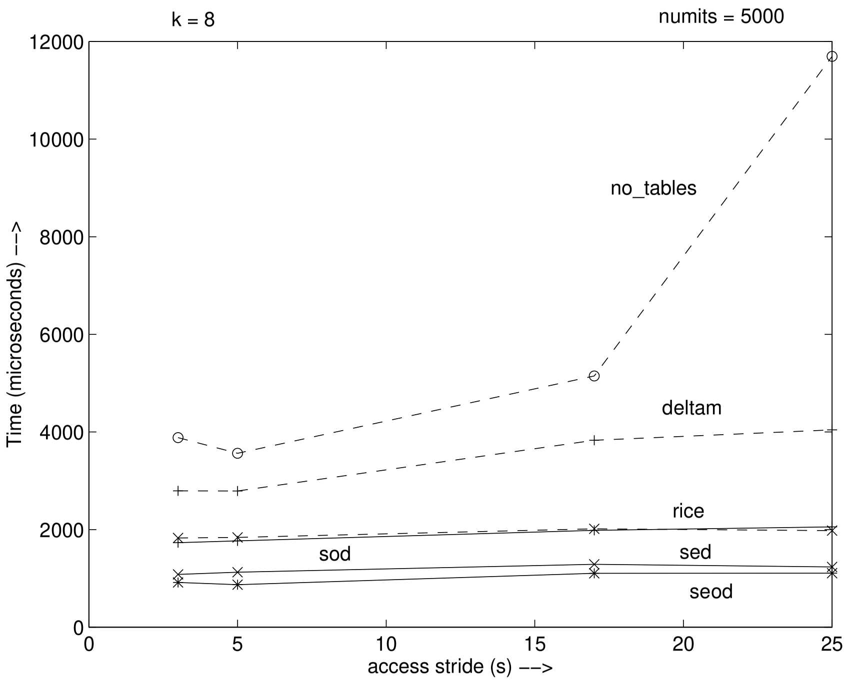
Processor 0				Processor 1				Processor 2				Processor 3			
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63

7

Computation of Start for $s > k$

Processor 0				Processor 1				Processor 2				Processor 3			
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
64	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
48	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127
128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143

MIV results – 1



MIV results – 2

k	s_2	SEOD	SED	SOD	RICE	DELTAM	NO TABLES
8	3	916	1082	1733	1829	2793	3885
	5	872	1128	1770	1839	2792	3565
	17	1104	1290	1986	2015	3830	5147
	25	1107	1234	2056	1979	4044	11695
32	3	917	1084	1770	1807	2758	3873
	5	939	1061	1779	1806	2791	3019
	17	1120	1280	2043	1994	3099	3375
	25	1283	1356	1999	2095	3087	6739
128	3	916	1065	1763	1793	2773	3892
	5	1129	1286	1895	1804	2772	3568
	17	1171	1368	1964	2008	2959	3633
	25	1225	1355	2021	2029	2931	6974

MIV Summary

- Methods to compute addresses in case of MIV
  - Outer loop
    - Basis Vectors
    - Compute start elements
    - Compute end + start elements
  - Inner Loop
    - Basis Vectors
    - k inner loop pattern tables
  - Interface
    - Which inner loop tables to use

Address Generation for Coupled Subscripts

```
      REAL A(M,N)
!HPF$ PROCESSORS PROC(p1, p2)
!HPF$ ALIGN A(i,j) WITH T(i,j)
!HPF$ DISTRIBUTE (CYCLIC(k1, k2)) ONTO Proc::  T

DO i = 0, ...
   A(i*s1 + l1, i*s2 + l2) = ...
ENDDO
```

Our Solution using Search or Hashing

Solution for address generation of coupled subscripts $\Longrightarrow$ find the common iterations among the two dimensions

- Pattern table size $=$ lcm $\left(\frac{p_1 k_1}{d_1}, \frac{p_2 k_2}{d_2}\right)$
- avoids solving $k_1 k_2$ linear Diophantine equations
- avoids global to local conversion
- space overhead for hashing, search (replication)
- theoretical complexity of the two algorithms much worse than $O(k_1 k_2 \log(k_1 k_2))$ but practically much better
- Closed forms for special cases: (BLOCK, BLOCK), (CYCLIC, CYCLIC), (BLOCK, CYCLIC), etc.

Coupled Subscripts Example

Processor Geometry = (4 x 3) **CYCLIC(2,4)** **s1 = 3, s2= 5**

Iteration table for first dimension for processor (0,0)

0	**3**	**8**	**11**	**16**	**19**	**24**	**27**	**32**	**35**	**40**	**43**	**48**	**51**	**...**

Iteration table for second dimension for processor (0,0)

0	**3**	**5**	**10**	**12**	**15**	**17**	**22**	**24**	**27**	**29**	**34**	**36**	**39**	**...**

Table construction for CS: 8×4, $s_1 = 5, s_2 = 3$

k_1	k_2	Rice	Search	Hash
4	4	23.4	24.6	22.6
4	8	30.7	30.9	30.8
8	4	34.6	32.1	31.8
8	8	58.5	28.7	31.4
16	4	63.8	34.5	31.3
16	8	86.8	36.8	33.5
8	16	72.7	34.6	29.1
16	16	140.2	41.0	33.8
32	4	105.2	43.7	35.1
32	8	158.1	43.0	37.8
8	32	134.7	38.0	36.0
32	32	463.0	57.5	45.1

k_1	k_2	Rice	Search	Hash
64	4	210.7	65.0	54.2
64	8	310.1	67.3	54.8
8	64	269.7	52.4	48.0
64	64	1842.6	82.1	76.2
128	4	410.2	106.9	95.9
128	8	612.5	103.5	121.2
8	128	526.6	80.0	73.0
128	128	6858.9	136.3	118.5
256	4	872.6	202.7	201.8
256	8	1300.7	196.3	195.2
8	256	1068.5	138.8	130.8
256	256	27765.3	258.6	254.8

Nextoffset data structure: Kennedy et al. (Rice)

Compute $start_1, start_2, end_1, end_2$
Compute a two-dimensional *Nextoffset* table
$a_1 \leftarrow start_1$
$a_2 \leftarrow start_2$
$o_1 \leftarrow start_1$ mod k_1
$o_2 \leftarrow start_2$ mod k_2
while $(a_1 \leq end_1)$ and $(a_2 \leq end_2)$ **do**
    $array[a_1, a_2] \leftarrow \cdots$
    $(a_1, a_2) \leftarrow (a_1, a_2) + \quad {}_2M[o_1, o_2]$
    $(a_1, a_2) \leftarrow$ *Nextoffset*$[o_1, o_2]$
endwhile

Node code using RESET table

```
Compute start_1, start_2, end_1, end_2
len ← length(  M)
for i = 0 to len − 2 do
    RESET[i] ← i + 1
endfor
RESET[len − 1] ← 0;
a_1 ← start_1
a_2 ← start_2
k ← 0
while (a_1 ≤ end_1) and (a_2 ≤ end_2) do
    array[a_1, a_2] ← ···
    (a_1, a_2) ← (a_1, a_2) +   M[k]
    (a_1, a_2) ← RESET[k]
endwhile
```


Nextoffset vs. RESET

- The *nextoffset* table used by Kennedy *et al.* can be much simplified by using a table where table$[i]$ = $(i+1)$mod length.
- If the inner loops are small, then the RESET table can be used on the expanded memory gap table as well.
- The size of the RESET table is a function of the length of the memory gap table, whereas the nextoffset table is always of length $k_1 k_2$.
- No need to recompute the reset table for cases where the lower bound of the array section changes.

Node code results for CS: 8×4, $s_1 = 5, s_2 = 3$

k_1	k_2	Rice	MOD	reset
4	4	591.9	758.9	400.2
4	8	609.1	770.2	419.6
8	4	610.1	794.5	423.1
8	8	649.4	835.8	455.6
16	4	652.0	874.0	465.2
16	8	736.1	965.6	540.4
8	16	717.0	908.0	540.1
16	16	873.9	1122.0	700.2
32	4	1499.2	983.4	576.6
32	8	890.2	1136.7	704.1
8	32	890.3	1082.5	697.3
32	32	1845.0	2032.0	1668.5

k_1	k_2	Rice	MOD	reset
64	4	1699.3	1151.8	768.2
64	8	2010.5	1511.7	1099.7
8	64	1229.8	1448.0	1039.4
64	64	5716.1	5847.5	5541.9
128	4	2098.9	1585.2	1374.0
128	8	2733.8	2213.6	1989.0
8	128	1872.0	2074.8	1687.1
128	128	20908.7	20991.8	20651.5
256	4	5800.2	2305.8	1937.5
256	8	6660.3	3579.4	3242.0
8	256	3254.3	3413.2	3072.2
256	256	85578.1	82371.3	82022.2

Summary and Work in Progress

- Summary: A spectrum of techniques for array accesses with complex subscript functions
  - Fast computation of start elements
  - Using tables is effective for multiple induction variables
  - Practical method for finding the address sequence for coupled subscripts
  - Efficient node code using RESET table for coupled subscripts
- Work in progress:
  - evaluation of execution strategies for coupled subscripts.
  - address generation for complex subscripts with two-level mapping.
  - code generation for general affine subscripts.