Distributed Memory Code Generation for Mixed Affine and Non-Affine Computations : Example

I. Original Code

Figure 1 represents the abstraction used in AMR applications. The solid line represents a box which uses a coarse grid, while the dashed lines represents boxes that use a finer grid and are colocated in physical space with boxes using a coarse grid. In the example code shown in Listing 1, the fine grid resolution is twice the coarse grid resolution along both directions, i.e., each coarse grid-point is associated with 4 fine grid-points. The resolution of the coarse grid is assumed to be same as the resolution of integer points of the underlying physical domain. The total number of boxes (fine and coarse) used in the computation is `nboxes`, with each box having an index between 0 and `nboxes-1`, both included. The arrays `fine_boxes` is of size equal to the number of fine boxes used in the computation and stores their indices. Similarly, array `coarse_boxes` is of size equal to the number of coarse boxes used in the computation and stores their indices.

The array `start_x` and `start_y` is of size equal to the number of boxes used in the computations, both coarse and fine. They store the integer co-ordinates of the lexicographically smallest point of each box, i.e., the co-ordinates of point E for the box shown in Figure 1. The array `end_x`, `end_y` are of the same size as `start_x`, `start_y` and store the co-ordinates of the lexicographically largest point of a box, i.e., point D. The values of ϕ and ϕ_{old} at grid points within a box are stored in arrays `phi` and `phi_old` respectively. These are 3D arrays, such that `phi[k][..][..]` holds the values of ϕ for the `k`-th box.

In the first annotated parallel loop at line 3, the values of `phi` at the coarse grid-points are updated using stencil operations with values of `phi_old` at the coarse grid-points used as input. The dotted line represent regions in surrounding boxes that are accessed during this operation. The array `phi_c_t` is used as a scratch pad to store values of `phi_old` from current and neighboring boxes used for the stencil operation. For each box, the information about patches of neighboring boxes accessed is maintained by storing the integer co-ordinates of the lexicographic minimum and maximum values of the patch in the arrays, `patch_start_y`, `patch_start_x`, `patch_end_y` and `patch_end_x`. For example, region within the north-neighboring box accessed is captured by storing the co-ordinates of point A in `patch_start_x` and `patch_start_y`, and the co-ordinates of point C are stored in `patch_end_x` and `patch_end_y`. The array `neighb` holds the index of the neighboring coarse boxes that contains this patch. Loops `i` and `j` at lines 10 and 11, iterate through the points within the patches. The position in `phi_old[neighb[p]][..][..]` of the required value from the neighboring box is obtained by subtracting the lexicographic minimum point of that box, i.e., point A, from the value of the loop iterators.

The second annotated loop at Line 20, updates values at fine-grid points with the values at coarse-grid points. Each fine box is assumed to be colocated in physical space with only one coarse box. The array `ftoc` stores the index of this coarse box for every fine box. The loops at line 21 and 22 iterate over the integer points bounded by the fine box. Since there are 4 fine grid-points for every coarse grid-points, statements from line 23 to 26 updates the value for each of them. In general, the value at a fine grid-point is updated using a stencil operation involving values at the coarse grid-point, with a different co-efficients used for each statement from line 23 to 26. Here, a simple copy is used. The points in the array `phi_old[fine_boxes[k]][..][..]` to be updated is indexed by subtracting the lexicographic minimum point of the fine box from the iterator values and multiplying it by the resolution of the finer grid along that dimension, i.e. 2 for both x and y. The position of grid points within `phi[ftoc[k]][..][..]` used on the right-hand side of these statements are obtained by subtracting from the iterator values, the lexicographic minimum of the coarse box.

II. Iteration Space Description

The first step in the code-generation scheme is to replace indirection array accesses within regular parts of the code with temporary variables. For each statement in Listing 1, the expressions replaced, the iteration

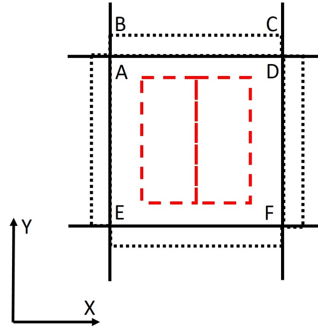

Fig. 1. Coarse and Fine Boxes used in AMR

```
1  #pragma parallel
2  #pragma temporary_array phi_c_t
3  for( k = 0; k < nc; k++ ){ ///Non-affine Iterator
4    for( i = 0; i < size_y[coarse_boxes[k]]; i++ ) ///Affine Iterator
5      for( j = 0; j < size_x[coarse_boxes[k]]; j++ ) ///Affine Iterator
6        phi_c_t[i+1][j+1] = phi_old[coarse_boxes[k]][i][j];
7
8    ///Non-affine Iterator
9    for( p = patches[coarse_boxes[k]]; p < patches[coarse_boxes[k]+1]; p++)
10     for( i = patch_start_y[p]; i < patch_end_y[p]; i++ ) ///Affine Iterator
11       for( j = patch_start_x[p]; j < patch_end_x[p]; j++ ) ///Affine Iterator
12         phi_c_t[i+1-start_y[coarse_boxes[k]]][j+1-start_x[coarse_boxes[k]]]
             = phi_old[neighb[p]][i - start_y[neighb[p]]][j - start_x[neighb[p]]];
13
14   for( i = 0; i < size_y[coarse_boxes[k]]; i++ ) ///Affine Iterator
15     for( j = 0; j < size_x[coarse_boxes[k]]; j++ ) ///Affine Iterator
16       phi[coarse_boxes[k]][i][j] += ( phi_c_t[i][j+1] + phi_c_t[i+1][j] + phi_c_t[i+1][j+2]
           + phi_c_t[i+2][j+1] - 4 * phi_c_t[i+1][j+1] - rho[coarse_boxes[k]][i][j] ) / 8.0;
17 }
18
19 #pragma parallel
20 for( k = 0; k < nf; k++) { ///Non-affine Iterator
21   for( i = start_y[fine_boxes[k]]; i < end_y[fine_boxes[k]]; i++ ) ///Affine Iterator
22     for( j = start_x[fine_boxes[k]]; j < end_x[fine_boxes[k]]; j++ ){ ///Affine Iterator
23      phi_old[fine_boxes[k]][(i-start_y[fine_boxes[k]])*2][(j-start_x[fine_boxes[k]])*2]
           = phi[ftoc[k]][i-start_y[ftoc[k]]][j-start_x[ftoc[k]]];
24      phi_old[fine_boxes[k]][(i-start_y[fine_boxes[k]])*2+1][(j-start_x[fine_boxes[k]])*2]
           = phi[ftoc[k]][i-start_y[ftoc[k]]][j-start_x[ftoc[k]]];
25      phi_old[fine_boxes[k]][(i-start_y[fine_boxes[k]])*2][(j-start_x[fine_boxes[k]])*2+1]
           = phi[ftoc[k]][i-start_y[ftoc[k]]][j-start_x[ftoc[k]]];
26      phi_old[fine_boxes[k]][(i-start_y[fine_boxes[k]])*2+1][(j-start_x[fine_boxes[k]])*2+1]
           = phi[ftoc[k]][i-start_y[ftoc[k]]][j-start_x[ftoc[k]]];
27     }
28 }
```

Listing 1. Original loop-nests of AMR

space and data access map for arrays `phi_old`, `phi` and `rho` are described below.

A. *Statement at Line 6*

Indirection array accesses expressions replaced by temporary variables for the statement :

- `p1 := coarse_boxes[k]`
- `p2 := size_y[coarse_boxes[k]]`
- `p3 := size_x[coarse_boxes[k]]`

Non Affine Iterators : `k`

Iteration Space

$$I_1 := \{(k, i, j) \mid k = kc \ \wedge \ 0 \leq i < p2 \ \wedge \ 0 \leq j < p3\} \tag{1}$$

Access to array `phi_old`

$$D_1 := \{(k, i, j) \rightarrow (l, a, b) \mid l = p1 \ \wedge \ a = i \ \wedge \ b = j\} \tag{2}$$

B. Statement at Line 12

Indirection array accesses expressions replaced by temporary variables for the statement :

- `p4 := patches[coarse_boxes[k]]`
- `p5 := patches[coarse_boxes[k]+1]`
- `p6 := patch_start_y[p]`
- `p7 := patch_end_y[p]`
- `p8 := patch_start_x[p]`
- `p9 := patch_end_x[p]`
- `p10 := start_y[coarse_boxes[k]]`
- `p11 := start_x[coarse_boxes[k]]`
- `p12 := start_y[neighb[p]]`
- `p13 := start_x[neighb[p]]`
- `p14 := neigh[p]`

Non Affine Iterators : `k,p`

Iteration Space

$$I_2 := \{(k, p, i, j) \mid k = kc \ \wedge \ p = pc \ \wedge \ p6 \leq i < p7 \ \wedge \ p8 \leq j < p9\} \tag{3}$$

Access to array `phi_old`

$$D_2 := \{(k, p, i, j) \rightarrow (l, a, b) \mid l = p14 \ \wedge \ a = i - p12 \ \wedge \ b = j - p13\} \tag{4}$$

C. Statement at Line 16

Indirection array accesses expressions replaced by temporary variables for the statement :

- `p1 := coarse_boxes[k]`
- `p2 := size_y[coarse_boxes[k]]`
- `p3 := size_x[coarse_boxes[k]]`

Non Affine Iterators : `k`

Iteration Space

$$I_3 := \{(k, i, j) \mid k = kc \ \wedge \ 0 \leq i < p2 \ \wedge \ 0 \leq j < p3\} \tag{5}$$

Access to array `phi`

$$D_3 := \{(k, i, j) \rightarrow (l, a, b) \mid l = p1 \ \wedge \ a = i \ \wedge \ b = j\} \tag{6}$$

Access to array `rho`

$$D_4 := \{(k, i, j) \rightarrow (l, a, b) \mid l = p1 \ \wedge \ a = i \ \wedge \ b = j\} \tag{7}$$

D. Statement at Line 23-26

Indirection array accesses expressions replaced by temporary variables for the statement :

- `p15 := fine_boxes[k]`
- `p16 := ftoc[k]`
- `p17 := start_y[fine_boxes[k]]`
- `p18 := start_x[fine_boxes[k]]`
- `p19 := end_y[fine_boxes[k]]`
- `p20 := end_x[fine_boxes[k]]`
- `p21 := start_y[ftoc[k]]`
- `p22 := start_x[ftoc[k]]`

Non Affine Iterators : `k`

Iteration Space

$$I_4 := \{(k,i,j) \mid k = kf \ \wedge \ p17 \le i < p19 \ \wedge \ p18 \le j < p20\} \tag{8}$$

Access to array `phi_old`

$$\begin{aligned} D_5 :=& \{(k,i,j) \to (l,a,b) \mid l = p15 \ \wedge \ a = 2*(i-p17) \ \wedge \ b = 2*(j-p18)\} \\ &\cup \{(k,i,j) \to (l,a,b) \mid l = p15 \ \wedge \ a = 2*(i-p17)+1 \ \wedge \ b = 2*(j-p18)\} \\ &\cup \{(k,i,j) \to (l,a,b) \mid l = p15 \ \wedge \ a = 2*(i-p17) \ \wedge \ b = 2*(j-p18)+1\} \\ &\cup \{(k,i,j) \to (l,a,b) \mid l = p15 \ \wedge \ a = 2*(i-p17)+1 \ \wedge \ b = 2*(j-p18)+1\} \end{aligned} \tag{9}$$

Access to array `phi`

$$D_6 := \{(k,i,j) \to (l,a,b) \mid l = p16 \ \wedge \ a = i - p21 \ \wedge \ b = j - p22\} \tag{10}$$

III. Inspector Code to compute the footprint

The inspector code generated using Algorithm 1 is shown in Listing 2.

```
1  for (k = 0; k < nc ; k++ )
2    if( get_home(loop_0,k) == myid ){
3      p1 = get_elem(array_coarse_box,k);
4      p2 = get_elem(array_size_y,get_elem(array_coarse_boxes,k));
5      p3 = get_elem(array_size_x,get_elem(array_coarse_boxes,k));
6
7      if( p2 >= 1 && p3 >= 1 ){
8        ///Record access to phi_old[coarse_boxes[k]]
9        p_outer = p1;
10       update_access(array_phi_old,p_outer);
11       ///Record access lexmin/max for all the inner dimension
12       ///lexmin(P_outer(D1(I1))
13       lexmin_dim1 = 0; lexmin_dim2 = 0;
14       update_lexmin(array_phi_old,p_outer,lexmin_dim1,lexmin_dim2);
15       ///lexmax(P_outer(D1(I1))
16       lexmax_dim1 = p2-1 ; lexmax_dim2 = p3-1;
17       update_lexmax(array_phi_old,p_outer,lexmax_dim1,lexmax_dim2);
18     }
19
20     for( p = get_elem(array_patches,get_elem(array_coarse_boxes,k)) ;
              p < get_elem(array_patches,get_elem(array_coarse_boxes,k)+1) ; p++ ){
21       p4 = get_elem(array_patches,get_elem(array_coarse_boxes,k));
22       p5 = get_elem(array_patches,get_elem(array_coarse_boxes,k)+1);
23       p6 = get_elem(array_patch_start_y,p);
24       p7 = get_elem(array_patch_end_y,p);
25       p8 = get_elem(array_patch_start_x,p);
26       p9 = get_elem(array_patch_end_x,p);
27       p10 = get_elem(array_start_y,get_elem(array_coarse_boxes,k));
28       p11 = get_elem(array_start_x,get_elem(array_coarse_boxes,k));
29       p12 = get_elem(array_start_y,get_elem(array_neighb,p));
30       p13 = get_elem(array_start_x,get_elem(array_neighb,p));
31       p14 = get_elem(array_neighb,p);
32
33       if( p7 >= p6 + 1 && p9 >= p8 + 1 ){
34         ///Record access to phi_old[neighb[p]]
35         p_outer = p14;
36         update_access(array_phi_old,p_outer);
37         ///Record access lexmin/max for all the inner dimension
38         ///lexmin(P_outer(D2(I2))
39         lexmin_dim1 = p6-p12; lexmin_dim2 = p8-p13;
40         update_lexmin(array_phi_old,p_outer,lexmin_dim1,lexmin_dim2);
41         ///lexmax(P_outer(D2(I2))
42         lexmax_dim1 = p7-p12-1; lexmax_dim2 = p9-p13-1;
43         update_lexmin(array_phi_old,p_outer,lexmax_dim1,lexmax_dim2);
44       }
45
46     if( p2 >= 1 && p3 >= 1 ){
47       ///Record access to phi[coarse_boxes[k]]
48       p_outer = p1;
49       update_access(array_phi,p_outer);
50       ///Record access lexmin/max for all the inner dimension
51       ///lexmin(P_outer(D3(I3))
```

```
52        lexmin_dim1 = 0 ; lexmin_dim2 = 0;
53        update_lexmin(array_phi,p_outer,lexmin_dim1,lexmin_dim2);
54        ///lexmax(P_outer(D3(I3))
55        lexmax_dim1 = p2-1 ; lexmax_dim2 = p3-1;
56        update_lexmax(array_phi,p_outer,lexmax_dim1,lexmax_dim2);
57
58        ///Record access to rho[coarse_boxes[k]]
59        p_outer = p1;
60        update_access(array_rho,p_outer);
61        ///Record access lexmin/max for all the inner dimension
62        ///lexmin(P_outer(D4(I3))
63        lexmin_dim1 = 0 ; lexmin_dim2 = 0;
64        update_lexmin(array_rho,p_outer,lexmin_dim1,lexmin_dim2);
65        ///lexmax(P_outer(D4(I3))
66        lexmax_dim1 = p2-1 ; lexmax_dim2 = p3-1;
67        update_lexmax(array_rho,p_outer,lexmax_dim1,lexmax_dim2);
68      }
69    }
70
71  for (k = 0; k < nf ; k++ )
72    if( get_home(loop_1,k) == myid ){
73
74      p15 = get_elem(array_fine_boxes,k);
75      p16 = get_elem(array_ftoc,k);
76      p17 = get_elem(array_start_y,get_elem(array_fine_boxes,k));
77      p18 = get_elem(array_start_x,get_elem(array_fine_boxes,k));
78      p19 = get_elem(array_end_y,get_elem(array_fine_boxes,k));
79      p20 = get_elem(array_end_x,get_elem(array_fine_boxes,k));
80      p21 = get_elem(array_start_y,get_elem(array_ftoc,k));
81      p22 = get_elem(array_start_x,get_elem(array_ftoc,k));
82
83      if( p19 >= p17 + 1 && p20 >= p18 + 1 ){
84        ///Record access to phi_old
85        p_outer = p15;
86        ///Record access lexmin/max for all the inner dimension
87        ///lexmin(P_outer(D5(I4))
88        lexmin_dim1 = 0; lexmin_dim2 = 0;
89        update_lexmin(array_phi_old,lexmin_dim1,lexmin_dim2);
90        ///lexmax(P_outer(D5(I4))
91        lexmax_dim1 = 2 * p19 - 2 * p17 - 1 ; lexmax_dim2 = 2 * p20 - 2 * p18 - 1;
92        update_lexmax(array_phi_old,lexmax_dim1,lexmax_dim2);
93
94        ///Record access to phi
95        p_outer = p16;
96        ///Record access lexmin/max for all the inner dimension
97        ///lexmin(P_outer(D6(I4))
98        lexmin_dim1 = p17 - p21; lexmin_dim2 = p18 - p22;
99        update_lexmin(array_phi,lexmin_dim1,lexmin_dim2);
100       ///lexmax(P_outer(D6(I4))
101       lexmax_dim1 = p19 - p21 - 1 ; lexmax_dim2 = p20 - p21 - 1;
102       update_lexmax(array_phi,lexmax_dim1,lexmax_dim2);
103     }
104   }
```

Listing 2. Inspector Code to compute footprint

IV. EXECUTOR CODE

The executor code generated using Algorithm 2 is shown in Listing 3.

```
1  ///Update ghosts in array phi_old with values at owners
2  communicate_reads(loop_0);
3  ///Initialize ghosts in array phi to 0.0;
4  init_write_ghosts(loop_0);
5  loop_0=0;
6  for (k = 0; k < nlocal_2; ++k) {
7    p1 = coarse_boxes_l[k];
8    p2 = size_y_l[coarse_boxes_l[k]];
9    p3 = size_x_l[coarse_boxes_l[k]];
10   for (i = 0; i < p2; ++i)
11     for (j = 0; j < p3; ++j)
12       phi_c_t[i+1][j+1] =
           phi_old_l[p1][i-lexmin_phi_old_dim1[p1]][j-lexmin_phi_old_dim2[p1]];
13
14   access_offset = access_phi_old[loop_0] - patches_l[coarse_boxes_l[k]];
```

```
15   for (p = patches_l[coarse_boxes_l[k]]; p < ub[k]; ++p) {
16     p4 = patches_l[coarse_boxes_l[k]];
17     p5 = ub[k];
18     p6 = patch_start_y_l[p+access_offset];
19     p7 = patch_end_y_l[p+access_offset];
20     p8 = patch_start_x_l[p+access_offset];
21     p9 = patch_end_x_l[p+access_offset];
22     p10 = start_y_l[coarse_boxes_l[k]];
23     p11 = start_x_l[coarse_boxes_l[k]];
24     p12 = start_y_l[neighb_l[p+access_offset]];
25     p13 = start_x_l[neighb_l[p+access_offset]];
26     p14 = neighb_l[p+access_offset];
27     for (i = p6 ; i < p7; ++i)
28       for (j = p8 ; j < p9; ++j)
29        phi_c_t[i+1-p10][j+1-p11] =
          phi_old_l[p14][i-p12-lexmin_phi_old_dim1[p14]][j-p13-lexmin_phi_old_dim1[p14]];
30   }
31   loop_0++;
32
33   for (i = 0; i < p2; ++i)
34     for (j = 0; j < p3; ++j)
35       phi_l[p1][i-lexmin_phi_dim1[p1]][j-lexmin_phi_dim2[p1]] +=
          (phi_c_t[i][j+1] + phi_c_t[i+1][j] + phi_c_t[i+1][j+2] + phi_c_t[i+2][j+1]
          - 4 * phi_c_t[i+1][j+1]
           - rho_l[p1][i-lexmin_rho_dim1[p1]][j-lexmin_rho_dim2[p1]] ) / 8.0;
36  }
37 ///Transfer partial contributions in ghosts of array phi to the owners
38 communicate_writes(loop_0);
39
40 ///Update ghosts in array phi with values at owners
41 communicate_reads(loop_1);
42 ///Initialize ghosts in array phi_old to 0;
43 init_write_ghosts(loop_1);
44 for (k = 0; k < nlocal_3; ++k) {
45  p15 = fine_boxes_l[k];
46  p16 = ftoc_l[k];
47  p17 = start_y_l[fine_boxes_l[k]];
48  p18 = start_x_l[fine_boxes_l[k]];
49  p19 = end_y_l[fine_boxes_l[k]];
50  p20 = end_x_l[fine_boxes_l[k]];
51  p21 = start_y_l[ftoc_l[k]];
52  p22 = start_x_l[ftoc_l[k]];
53  for (i = p17 ; i < p19 ; i++ )
54   for (j = p18 ; j < p20 ; j++ ){
55    phi_old_l[p15][2*(i-p17)-lexmin_phi_old_dim1[p15]][2*(j-p18)-lexmin_phi_old_dim2[p15]]
      = phi_l[p16][i - p21 - lexmin_phi_dim1[p16]][j - p22 - lexmin_phi_dim2[p16]];
56    phi_old_l[p15][2*(i-p17)+1-lexmin_phi_old_dim1[p15]][2*(j-p18)-lexmin_phi_old_dim2[p15]]
      = phi_l[p16][i - p21 - lexmin_phi_dim1[p16]][j - p22 - lexmin_phi_dim2[p16]];
57    phi_old_l[p15][2*(i-p17)-lexmin_phi_old_dim1[p15]][2*(j-p18)+1-lexmin_phi_old_dim2[p15]]
      = phi_l[p16][i - p21 - lexmin_phi_dim1[p16]][j - p22 - lexmin_phi_dim2[p16]];
58  phi_old_l[p15][2*(i-p17)+1-lexmin_phi_old_dim1[p15]][2*(j-p18)+1-lexmin_phi_old_dim2[p15]]
      = phi_l[p16][i - p21 - lexmin_phi_dim1[p16]][j - p22 - lexmin_phi_dim2[p16]];
59   }
60  }
61 ///Transfer partial contributions in ghosts of array phi_old to the owners
62 communicate_writes(loop_1);
```

Listing 3. Executor Code