Overview

- Using a combination of compile-time and run-time analysis to optimize memory access and load balancing

- Targeted for HPF applications on distributed memory machines where data distribution and access parameters are not known at compile time

- Compiler extensions insert calls to a library of routines that can optimize data access at runtime
**Multigrid**

- **Multigrid applications**
  - Use a combination of coarse and fine meshes to accelerate computation via approximations
  - Used for solving PDEs
  - Mesh resolution and nesting can vary at runtime
  - Data distribution may be necessary within a meshes and when propagating values between meshes

**Multiblock**

- **Multiblock applications**
  - Sets of meshes (blocks) with non-uniform structure
  - Adjacent blocks may have different mesh sizes
  - Outer loop performs time step
  - Data distribution both within and between blocks
  - Data distribution dependent on the object’s mesh
Runtime library optimizations

- Multiblock Parti library
- Dynamic data distribution
  - Regular section moves
- Partitioning loops via symbolic loop bounds and strides

Runtime primitives

- Communication schedule
  - Schedule describes data motion between processors
  - Goal is to look at the data distribution at runtime and build a schedule that optimizes data communication
  - Schedules can be reused for similar data distributions
- Data movement
Data distribution

- Regular section move
  - For all loops with array assignments where loop bounds and strides not known at compile time
  - Schedule determines data elements sent and received by each processor
  - Moving array elements between processors i.e. From one distributed array to another
  - Regular_section_copy_sched(…)
- Often only ghost/overlap cells between adjacent blocks/meshes need to copied
  - Overlap_cell_fill_sched(…)

Data distribution con’t

- Loop partitioning → handling symbolic loop bounds and strides
  - Owner computes rule
    - Loop iteration performed by process owning LHS array element
  - Loop bound transformations
    - Local_lower_bound()/Local_upper_bound() used to transform loop bounds
  - Mapping for local indices
    - Local_to_global/global_to_local
Compiler support

- Extensions to accommodate operations in the runtime library
- Extend processor abstraction to support subspaces
  - PROCESSORS P(N)
  - PSUBSPACE P1 IS P(UPPER:LOWER)
- Extend Align to specify border/ghost cells
  - ALIGN A(i,j) WITH T(i:2:3,j:2:3)

Communication patterns

- Methodology for analyzing forall loops and performing data moves when necessary
- Case I
  - Array A,B aligned to different template
  - No information about relationship
- Case II
  - Array A,B aligned to same template
  - A,B same size and shape
- Case III
  - Array A,B aligned to same template
  - Different loop bounds, strides
Communication patterns

- Look at loop bounds/strides and classify loops as follows
  - Not requiring any communication
  - Can be handled by filling overlap/ghost cells
  - Requiring regular section moves

<table>
<thead>
<tr>
<th>L.H.S. Expression</th>
<th>R.H.S. Expression</th>
<th>Regular Section Move Required</th>
<th>Overlap Cell Fill Required</th>
</tr>
</thead>
<tbody>
<tr>
<td>A(i, j)</td>
<td>B(i, j)</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>A(i+k, j)</td>
<td>B(i-k, j)</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>(A[i+k, j])</td>
<td>B(i-k, j)</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>A(i+k, j)</td>
<td>B(i+1+k, j)</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>A(i+k, j)</td>
<td>B(i-1+k, j)</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

C  **ORIGINAL HPF CODE**

C  **Arrays A, B are distributed identically**

fo(a (i = 1:100/2, j = 1:100) A(i, j) = B(2*i, j))

C  **TRANSFORMED CODE**

    NumSrcDim = 2    NumDestDim = 2
    SrcDim(1) = 2    DestDim(1) = 1
    SrcDim(2) = 1    DestDim(2) = 2
    SrcLo(1) = 2    DestLo(1) = 1
    SrcLo(2) = 1    DestLo(2) = 1
    SrcHi(1) = 100  DestHi(1) = 100
    SrcHi(2) = 100  DestHi(2) = 50
    SrcStr(1) = 2   DestStr(1) = 2
    SrcStr(2) = 2   DestStr(2) = 1

    Sched = Regular, Section, Move, Sched(DAD, DAD, NumSrcDim, NumDestDim, SrcDim, SrcLo, SrcHi, SrcStr, DestDim, DestLo, DestHi, DestStr)

    Call Data_Move(D_Sched, A)

  **Fig. 3. Regular section move example.**
Overhead

- Cost of copying (I vs. II) < 5%
- Scheduling time is small
  - Especially true on larger problems with regular access patterns

Case studies

- Comparing compiler optimized with hand optimized codes
- Both using same runtime library routines
- Compiler optimized within 10-20% of hand optimized
More results

- Version I rebuilds schedule during each loop iteration
- Increased communication from distributing blocks over entire process space

<table>
<thead>
<tr>
<th>No. of</th>
<th>Compiler Version I</th>
<th>Compiler Version II</th>
<th>Compiler Version III</th>
<th>Compiler Version IV</th>
<th>Hand</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pelis</td>
<td>13.45</td>
<td>7.63</td>
<td>7.41</td>
<td>7.33</td>
<td>6.79</td>
</tr>
<tr>
<td>8</td>
<td>15.81</td>
<td>4.78</td>
<td>4.58</td>
<td>4.54</td>
<td>4.19</td>
</tr>
<tr>
<td>16</td>
<td>18.72</td>
<td>2.83</td>
<td>2.71</td>
<td>2.63</td>
<td>2.39</td>
</tr>
<tr>
<td>32</td>
<td>8.03</td>
<td>1.85</td>
<td>1.79</td>
<td>1.66</td>
<td>1.47</td>
</tr>
</tbody>
</table>

Version I: Random library does not reuse scheduler
Version II: Random library reused schedules
Version III: Schedule reused by hand
Version IV: Loop truchre reused within a procedure

Fig. 8. Effects of various optimizations (sec).

<table>
<thead>
<tr>
<th>No. of Processes</th>
<th>Blocks Mapped Entire Processor Space</th>
<th>Blocks Mapped Eligible Processor Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>8.99</td>
<td>7.39</td>
</tr>
<tr>
<td>8</td>
<td>5.14</td>
<td>4.74</td>
</tr>
<tr>
<td>16</td>
<td>3.24</td>
<td>2.83</td>
</tr>
<tr>
<td>32</td>
<td>2.41</td>
<td>1.87</td>
</tr>
</tbody>
</table>

Fig. 9. Effect of data distribution.