An Integrated Runtime and Compile-Time Approach for Parallelizing Structured and Block Structured Applications

Gagan Agrawal, Alan Sussman, Joel Saltz
Presented by Sam Angiuoli

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 course 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
  - Column 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 be 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>Loop bound classification</th>
<th>Regular section move</th>
<th>Requiring regular section moves</th>
</tr>
</thead>
<tbody>
<tr>
<td>NewDim(.), NewDim(.), NewDim(.)</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>NewDim(.), NewDim(.), NewDim(.)</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>NewDim(.), NewDim(.), NewDim(.)</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>NewDim(.), NewDim(.), NewDim(.)</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Case IV</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

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

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>Loop bound classification</th>
<th>Regular section move</th>
<th>Requiring regular section moves</th>
</tr>
</thead>
<tbody>
<tr>
<td>NewDim(.), NewDim(.), NewDim(.)</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>NewDim(.), NewDim(.), NewDim(.)</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>NewDim(.), NewDim(.), NewDim(.)</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>NewDim(.), NewDim(.), NewDim(.)</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Case IV</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Overhead

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

![Graph showing overhead comparison between versions I and II.]

Case studies

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

![Table comparing performance of compiler and hand optimized codes.]

More results

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

![Graph showing increased communication.]

![Table showing effect of various optimizations.]

Fig. 7: Comparison of overhead performance (sec).