Efficient Run-Time Support for Irregular Block-Structured Applications

KeLP

S. Fink, S. Baden
Presented By Nicholas Kuilema
Runtime Parallelization

- Not a Runtime Environment like a JVM
- Dynamically performing the parallel separation based on the input when the program is run
- Uses details not available at compile time
KeLP - Goals

• Make Programming Easier
  • Irregular data sets
  • Data distribution

• Minimal sacrifice in performance
KeLP - Kernel Level Parallelism

• C++ Library
• Object Oriented
• Uses MPI for communication
• Best used on block structured applications
Key Ideas

• Data orchestration model
  Separates communication description and implementation

• Structural abstraction
  Separates data description and storage implementation
Programming Model

- Single Logical Thread
- Periodic SPMD loops

```plaintext
set up distributed data structures
do i = 1, n\text{iters}
   perform data motion
   \text{for all(...)}
   \quad \text{SMPD computation}
   \text{end for all}
end
```
• Structural abstraction built off of LPARX
• LPARX allows block copy from inside for_all
• LPARX has only one-sided communication
• Changes allow KeLP to be 40-70% faster
Data Layout Abstraction

- From LPARX
  - Point, Region, Grid, XArray
- Adds
  - MotionPlan
  - FloorPlan
  - Mover
Motion

- Motion Plans: Programmer supplied logical operations for memory relationships
- Movers: Interpret the plan down to specifics for the hardware or lower level API
FIG. 4. The MotionPlan encodes a set of block copy operations between grids.
### Table 1
A Brief Synopsis of the KeLP Data Types

<table>
<thead>
<tr>
<th>Geometric structural abstractions</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Name</strong></td>
<td><strong>Definition</strong></td>
<td><strong>Interpretation</strong></td>
</tr>
<tr>
<td>PointD</td>
<td>(\langle \text{int } i_0, \text{int } i_1, \ldots, \text{int } i_{D-1} \rangle)</td>
<td>A point in (\mathbb{Z}^D)</td>
</tr>
<tr>
<td>RegionD</td>
<td>(\langle \text{PointD } l, \text{PointD } h \rangle)</td>
<td>A rectangular subset of (\mathbb{Z}^D)</td>
</tr>
<tr>
<td>FloorPlanD</td>
<td>Array of (\langle \text{RegionD } R, \text{int } p \rangle)</td>
<td>A set of regions, each assigned to a processor (p)</td>
</tr>
<tr>
<td>MotionPlanD</td>
<td>List of (\langle \langle \text{int } f, \text{RegionD } R_f \rangle, \langle \text{int } t, \text{RegionD } R_t \rangle \rangle)</td>
<td>Block-structured communication pattern between two FloorPlans</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Data types that interpret abstractions</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Name</strong></td>
<td><strong>Description</strong></td>
<td></td>
</tr>
<tr>
<td>GridD</td>
<td>A multidimensional array whose index space is a RegionD</td>
<td></td>
</tr>
<tr>
<td>XArrayD</td>
<td>An array of GridDs; structure represented by a FloorPlanD</td>
<td></td>
</tr>
<tr>
<td>MoverD</td>
<td>Object that atomically performs the data motion pattern described by a MotionPlan</td>
<td></td>
</tr>
</tbody>
</table>
FIG. 3. The XArray is a coarse-grained distributed array of blocks of data, whose structure is described by a FloorPlan. The blocks may have different sizes and each is assigned to a single address space.
Data Layout Abstraction

- Adds Region Calculus

<table>
<thead>
<tr>
<th>Table 2</th>
<th>Region Calculus Operations Used in the Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Operation</strong></td>
<td><strong>Interpretation</strong></td>
</tr>
<tr>
<td><code>extents(Region R, int i)</code></td>
<td>Length of Region $R$ along the $i$th axis</td>
</tr>
<tr>
<td><code>shift(Region R, Point P)</code></td>
<td>Translation of Region $R$ by the vector $P$</td>
</tr>
<tr>
<td>Region $R \cap$ Region $S$</td>
<td>Geometric intersection of Regions $R$ and $S$</td>
</tr>
<tr>
<td><code>grow(Region R, Point P)</code></td>
<td>Region $R$ padded with $P(i)$ cells on $i$th axis</td>
</tr>
</tbody>
</table>
FillPatch

**FIG. 5.** (a) Pseudocode to generate a `fillpatch` MotionPlan $M$ to fill in ghost cells for XArray $x$. (b) The dark shaded regions represent ghost regions that are copied into the central Grid.
FIG. 8. (a) Block partitioning into 16 regions. (b) Each grid is padded with a layer of ghost cells to hold boundary conditions. (c) Dependencies that must be satisfied to refresh the ghost cells.
void fillGhost(XArray2<Grid2<double> > & X)
{
(1)    MotionPlan2 M;
(2)    for_1(i,X)
(3)        Region2 inside = grow(X(i).region(), -1);
(4)    for_1(j,X)
(5)        if (i != j) {
(6)            M.CopyOnIntersection(X,i,X,j,inside);
(7)        }
(8)    end_for
(9)    end_for
(10) Mover2<Grid2<double>, double> DM(X,X,M);
(11) DM.execute();
}

FIG. 9. FillGhost( ) function for code of Fig. 6.
## Performance vs MPI

### Table 3
Comparison of KeLP Performance with Three MPI Reference Codes on an IBM SP2

<table>
<thead>
<tr>
<th>Code</th>
<th>KeLP performance (MFLOPS)</th>
<th>MPI code performance (MFLOPS)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>8 nodes</td>
<td>16 nodes</td>
</tr>
<tr>
<td>NAS-MG Class B</td>
<td>311</td>
<td>558</td>
</tr>
<tr>
<td>NAS-FT Class A</td>
<td>111</td>
<td>203</td>
</tr>
<tr>
<td>SUMMA</td>
<td>1231</td>
<td>2452</td>
</tr>
<tr>
<td>Normalized KeLP running time (normalized so MPI running time = 1.0)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NAS-MG Class B</td>
<td>0.91</td>
<td>0.86</td>
</tr>
<tr>
<td>NAS-FT Class A</td>
<td>0.91</td>
<td>1.10</td>
</tr>
<tr>
<td>SUMMA</td>
<td>1.02</td>
<td>1.00</td>
</tr>
<tr>
<td>Percentage of time spent communicating in KeLP version</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NAS-MG Class B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NAS-FT Class A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SUMMA</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Performance vs LPARX

**Performance vs LPARX**

**IBM SP2**

- LPARX
- Kelp

**Intel Paragon**

- LPARX
- Kelp

Bars represent different components:
- Interlevel Communication
- Intralevel Communication
- Load Imbalance
- Computation

**Number of Processors**

- IBM SP2:
  - 4, 8, 16

- Intel Paragon:
  - 8, 16, 32
KeLP Conclusion

- Easier to program than MPI
- Good Performance
- Abstraction allows re-targeting
- No development since 2002
- Download link prompts for registration
An Integrated Runtime and Compile-Time Approach for Parallelizing Structured and block Structured Applications

G. Agrawal, A. Sussman, J. Saltz
Presented by Nicholas Kuilema
Goals

• Similar to KeLP
• Provide good performance
• Easy to program/maintain
• Focuses on distributed memory machines
• Multiple Meshes/Grids
• Regions vary in size/resolution
• Optimal distribution may depend on input
Multiblock Applications

- Irregularly coupled regular meshes
Multigrid

- Multiple meshes at different resolutions
- Operations coarsen/refine meshes
- Meshes may span processors
- Run time data needed for ideal distribution
Runtime Support

• Library called Multiblock Parti

• Performs three tasks
  • Distributed Array Descriptor (DAD)
  • Handles communication
  • Partitions work
Runtime

- Create a communication schedule (once)
  - Description of data motion
  - Look at dynamic actions to optimize communication
  - Reusable amortizes cost
- Do the communication (multiple times)
Data movement

• Regular section move
  • For all loops where strides/bounds are determined at runtime
  • A single array on the right hand side
  • Can be thought of as a move

• Overlap cell fill
  • Ghost cells at borders between sections
Data movement

- global_to_local
- local_to_global
- Owner computes
- Local_Lower_Bound
- Local_Upper_Bound
$PROCESSORS P(N)

$PSUBSPACE P1 IS P(LB:UB)

$TEMPLATE T(100,100)
$DISTRIBUTE T(BLOCK,BLOCK) ONTO P1

$DIMENSION A(105,105)
$ALIGN A(i,j) WITH T(i:2:3, j:2:3)
• Case 1: Arrays A & B are aligned to different templates or no information about relationship

• Case 2: Arrays A & B are aligned to the same template and have identical shape & size

• Case 3: Arrays A & B are aligned to the same template, but differ in stride, offset or index
For each case determine if:

- No communication needed
- Can solve with filling overlap cells
- Requires regular section moves

Case 1: Bulk moves must be used
• Case 2: Do communication if any of these

  • Loop index permutation
  • In dimension J the strides are different
  • Non-constant offsets
  • Offsets are different and unknown at compile
  • A copy but the variables are based on another dimension
Case 3:

<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, j)</td>
<td>B(j, i)</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>(A(2*i, j)</td>
<td>B(j, i)</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>A(2*i, j)</td>
<td>B(j, i + 3)</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>A(2*i, j)</td>
<td>B(j - 1, i)</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>
Workload Partitioning

C  ORIGINAL HPF CODE
C  Arrays A, B are distributed identically
    forall (i = 1:100:2, j = 1:50) A(i,j) = B(2*j,i)

C  TRANSFORMED CODE
NumSrcDim = 2  NumDestDim = 2
SrcDim(1) = 2  DestDim(1) = 1
SrcDim(2) = 1  DestDim(2) = 2
SrcLos(1) = 2  DestLos(1) = 1
SrcLos(2) = 1  DestLos(2) = 1
SrcHis(1) = 100  DestHis(1) = 100
SrcHis(2) = 100  DestHis(2) = 50
SrcStr(1) = 2  DestStr(1) = 2
SrcStr(2) = 2  DestStr(2) = 1
Sched = Regular_Section_Move_Sched(DAD, DAD, NumSrcDim, NumDestDim,
                                   SrcDim, SrcLos, SrcHis, SrcStr,
                                   DestDim, DestLos, DestHis, DestStr)
Call Data_Move(B, Sched, A)
Overhead of Primitives

- Scheduling time very small
- Overhead < 5%
Multiblock performance

Within 20%

**One Block: 49 × 9 × 9 Mesh (50 Iterations)**

<table>
<thead>
<tr>
<th>Number of Processors</th>
<th>Compiler Parallelized</th>
<th>Hand Parallelized F90</th>
<th>Hand Parallelized F77</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>6.99</td>
<td>6.88</td>
<td>6.20</td>
</tr>
<tr>
<td>8</td>
<td>4.17</td>
<td>4.06</td>
<td>4.00</td>
</tr>
<tr>
<td>16</td>
<td>2.47</td>
<td>2.35</td>
<td>2.28</td>
</tr>
<tr>
<td>32</td>
<td>1.55</td>
<td>1.45</td>
<td>1.41</td>
</tr>
</tbody>
</table>

*Fig. 5. Performance comparison for small mesh, one block (sec).*

**Two Blocks: 49 × 17 × 9 Mesh (50 Iterations)**

<table>
<thead>
<tr>
<th>Number of Processors</th>
<th>Compiler Parallelized</th>
<th>Hand Parallelized F90</th>
<th>Hand Parallelized F77</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>7.49</td>
<td>6.69</td>
<td>6.17</td>
</tr>
<tr>
<td>16</td>
<td>4.64</td>
<td>4.07</td>
<td>4.03</td>
</tr>
<tr>
<td>32</td>
<td>2.88</td>
<td>2.32</td>
<td>2.30</td>
</tr>
</tbody>
</table>

*Fig. 6. Performance comparison for larger mesh, two blocks (sec).*
### Multigrid performance

Within 10%

<table>
<thead>
<tr>
<th>No. of Proc.</th>
<th>Compiler: First Iteration</th>
<th>Compiler: Per-subsequent Iteration</th>
<th>By Hand: First Iteration</th>
<th>By Hand: Per-subsequent Iteration</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>4.80</td>
<td>2.29</td>
<td>4.60</td>
<td>2.14</td>
</tr>
<tr>
<td>16</td>
<td>3.84</td>
<td>1.38</td>
<td>3.41</td>
<td>1.35</td>
</tr>
<tr>
<td>32</td>
<td>3.03</td>
<td>0.95</td>
<td>2.48</td>
<td>0.88</td>
</tr>
</tbody>
</table>

Fig. 7. Semicoarsening multigrid performance (sec).
### Compiler Optimizations

**Two Blocks: 49 × 9 × 9 Mesh (50 iterations)**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>4</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>15.51</td>
<td>4.78</td>
<td>4.58</td>
<td>4.54</td>
<td>4.19</td>
</tr>
<tr>
<td>16</td>
<td>11.72</td>
<td>2.85</td>
<td>2.71</td>
<td>2.62</td>
<td>2.39</td>
</tr>
<tr>
<td>32</td>
<td>8.01</td>
<td>1.85</td>
<td>1.79</td>
<td>1.66</td>
<td>1.47</td>
</tr>
</tbody>
</table>

Version I: Runtime Library does not save schedules  
Version II: Runtime Library saves schedules  
Version III: Schedule reuse implemented by hand  
Version IV: Loop bounds reused within a procedure

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

Disjoint improves 10-25%

<table>
<thead>
<tr>
<th>No. of Processors</th>
<th>Blocks Mapped Entire Processor Space</th>
<th>Blocks Mapped Disjoint Processor Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>8.99</td>
<td>7.59</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>
Conclusion

- Reasonable performance
- Easier than hand writing MPI
- Helped bring about KeLP

- Questions?