Efficient Run-time Support for Irregular Block-Structured Applications
By Stephen J. Fink and Scott B. Baden and Scott R. Kohn

Presented for your delectation by Asad B. Sayeed
Background

- Main type of application considered: scientific numerical methods.
- These applications often use structured irregular representations to improve accuracy.
  - Difficult to implement.
  - Cause unpredictable/irregular communication patterns, impeding performance optimization.
- Goal: assist programmer in arranging parallelism so the data layout and distribution best exploit memory arrangements.
Kernel Lattice Parallelism

- KeLP: Kernel Lattice Parallelism.
  - Library for higher level abstractions for managing data layout and data motion.
  - Applications with dynamic block structures: uniform rectangular data arrays with irregular data motion.
  - Geometric programming abstractions represent data layout and motion patterns
  - Data orchestration model: separate description of motion patterns from interpretation/implementation.
  - Structural abstraction: separate structure of data from storage.
Kernel Lattice Parallelism

- System implemented on top of MPI in C++.
- Data orchestration implemented via MotionPlans and Movers.
  - Programmers define MotionPlans to schedule communication via geometric operations on memory structure.
  - Movers interpret the plans so as to conform to the hardware architecture and other application-specific issues.
Programming Model

- Programs begin with a single (logical) control thread.
- for_all loop iterations: each one executes independently on one SPMD process.
- Storage model: distribute each block of data to its own logical address space, one space per processor.
- Little compiler automation, even for consistency.
  - Programmer explicitly describes data decomposition and also data motion (via block copy operations).
Data Layout Abstractions

- Four core data decompositions abstractions: Point, Region, Grid, XArray, inherited from KeLP's predecessor LPARX. KeLP innovation: FloorPlans.
  - Point: represents point in n-dim space.
  - Region: rectangular subset of Points..
  - Grid: array of data indexed by Region.
  - XArray: array of Grids of different (irregular) shape.
  - FloorPlans: array of Regions representing processor assignments for XArray.
Data Layout Abstractions

• Regions are constructed by Region calculus.

<table>
<thead>
<tr>
<th>Table 2</th>
<th>Region Calculus Operations Used in the Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operation</td>
<td>Interpretation</td>
</tr>
<tr>
<td>extents(Region $R$, int $i$)</td>
<td>Length of Region $R$ along the $i$th axis</td>
</tr>
<tr>
<td>shift(Region $R$, Point $P$)</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>grow(Region $R$, Point $P$)</td>
<td>Region $R$ padded with $P(i)$ cells on $i$th axis</td>
</tr>
</tbody>
</table>

• XArrays and FloorPlans:

**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 Motion Abstractions

- **MotionPlan**
  - Data motion pattern defined over Xarrays.
  - Specified by programmer as set of array copy operations built via Region calculus.
  - Let G, H be Grids; let R, S be Regions.
    \{G \text{ on } R \rightarrow H \text{ on } S\} \text{ means copy index region } R \text{ from } G \text{ to region } S \text{ from } H.

- **Mover**: analyzes MotionPlan and performs movement.
  - Programmer can extend the Mover class to represent various communication operations.
Data Motion Abstractions

- MotionPlans illustrated:

*FIG. 4. The MotionPlan encodes a set of block copy operations between grids.*
Data Layout and Data Motion

- Summary of classes:

<table>
<thead>
<tr>
<th>Table 1</th>
<th>A Brief Synopsis of the KeLP Data Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Definition</td>
</tr>
<tr>
<td>---------</td>
<td>-----------------------------</td>
</tr>
<tr>
<td>PointD</td>
<td>(\langle \text{int } I_0, \text{int } I_1, \ldots, \text{int } I_{D-1} \rangle)</td>
</tr>
<tr>
<td>RegionD</td>
<td>(\langle \text{PointD } I, \text{PointD } h \rangle)</td>
</tr>
<tr>
<td>FloorPlanD</td>
<td>Array of (\langle \text{RegionD } R, \text{int } p \rangle)</td>
</tr>
<tr>
<td>MotionPlanD</td>
<td>List of (\langle\langle \text{int } f, \text{RegionD } R_i \rangle, \langle \text{int } t, \text{RegionD } R_i \rangle\rangle)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Name</th>
<th>Data types that interpret abstractions</th>
</tr>
</thead>
<tbody>
<tr>
<td>GridD</td>
<td>A multidimensional array whose index space is a RegionD</td>
</tr>
<tr>
<td>XArrayD</td>
<td>An array of GridDs; structure represented by a FloorPlanD</td>
</tr>
<tr>
<td>MoverD</td>
<td>Object that atomically performs the data motion pattern described by a MotionPlanD</td>
</tr>
</tbody>
</table>
Simple Data Motion Example

- **fillpatch**: fills in ghost cells from logically overlapping grids.

- Code and example of irregular XArray.

---

```
BuildFillpatch(XArray X, MotionPlan M)
begin
    for each i ∈ X
        I = grow(X(i), -1)
        for each j ∈ X
            if (i != j) {
                R = I ∩ X(j)
                M.Copy(X, i, R, X, j, R)
            }
        end for
    end for
end
```

**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.
Bigger Ghost Cells Example

- Elliptic PDE solver:

  (1) Region2 domain(1,1,N,N);
  (2) Processors2 P;
  (3) Decomposition2 T(domain);
  (4) T.distribute(BLOCK,BLOCK,P);
  (5) for_1(i,T)
  (6)     T.setregion(i,grow(T(i),1));
  (7) end_for
  (8) XArray2< Grid2<double> > U(T);
  (9) InitGrid(U);
  (10) int RedBlack = 0
  (11) for (int k=0 ; k<NITERS*2; k++) {
  (12)     fillGhost(U);
  (13)     for_all(i,U)
  (14)        sweep(U(i),RedBlack);
  (15)     end_for_all
  (16)     RedBlack = 1 - RedBlack;
  (17) }

  FIG. 7. Main procedure for red–black Gauss–Seidel example.

- Region2, etc are 2D arrays.
Bigger Ghost Cells Example

- Elliptic PDE FloorPlan:

- for_1 vs for_all -> current thread vs distributed
- The for_1 loop does initial ghost cell padding.
- Sweep: performs iteration, probably in Fortran.
Bigger Ghost Cells Example

- fillGhost function: central to parallelism.

```c
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();
}```

- Mostly just implements fillPatch from before.
  - Last two lines perform the movement.
  - We're recomputing ghost cells each time: not normal.
**Implementation Issues**

- KeLP predecessor: LPARX.
  - LPARX allowed asynchronous one-sided communication: creates barriers for process state global synchronization.
  - KeLP: bans copy operations from for_all loops, eliminating this problem; i.e., only for_1 loops perform copies. Each process stores relevant portion of movement plan.

- Mover: implemented via nonblocking MPI send
  - In and out buffers allocated to each process.
  - Receives data while it waits for sends to finish.
Performance

- Comparison to MPI.

- Three benchmarks involving heavy matrix computation: NAS-FT, NAS-MG, SUMMA.

<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>
</tbody>
</table>

- Normalized KeLP running time (normalized so MPI running time = 1.0)

<table>
<thead>
<tr>
<th>Code</th>
<th>Normalized KeLP running time</th>
<th>Percentage of time spent communicating in KeLP version</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAS-MG Class B</td>
<td>0.91</td>
<td>22</td>
</tr>
<tr>
<td>NAS-FT Class A</td>
<td>0.91</td>
<td>15</td>
</tr>
<tr>
<td>SUMMA</td>
<td>1.02</td>
<td>23</td>
</tr>
</tbody>
</table>
Performance

- Adaptive multigrid: lda3d
  - Eigenvalue solver from “ab initio” materials science.
  - Highly irregular communication.

FIG. 10. Performance comparison for adaptive multigrid eigenvalue solver on the (a) IBM SP2 and (b) Intel Paragon.
Performance

- **jacobi3d**: tuning communication performance.
  - Three KeLP versions and one hand-coded version.
  - KeLP versions vary by ghost cell communication arrangements.
Related Work

• KeLP: structural abstraction from LPARX combined with inspector/executor communication analysis.

• Other structural abstraction implementation: BOXLIB, DAGH. More specialized.

• Inspector/executor appears in Multiblock PARTI, which does not allow irregular block decompositions—doesn't have same level of structural abstraction.

• Number of other related applications.
Conclusion

- Structural abstractions hide some of the dirty work required for efficient communication within irregular block decompositions.

- Despite KeLP being a high-level abstraction over MPI, performs very favorably compared to MPI.

- Inspector/executor paradigm (MotionPlans vs Movers) allows retargetting to various situations.

*Kernel Lettuce Decomposition, more widely eaten than KeLP,*