Linear Algebra Programming Motifs

John G. Lewis

Cray Inc. (retired)

March 2, 2011

Programming Motifs 1, 2 & 9

- Dense Linear Algebra
- Graph Algorithms (and Sparse Matrix Reordering)

Features Used in Linear Algebra and Graph Algorithms

- Global Addressing Model with Affinity (domains and distributions for arrays on large scale systems)
- Named Index Sets (blocking via domains and iterators)
- Data and Task Parallel constructs (expressing parallelism)
- Generic Programming and Object Orientation (data-structure independent code for sparse matrices and graphs)
- Sparse Sets (sparse and associative domains)

Global Address Space

- For simplicity – parallel matrix transpose $C = \beta C + A^T$
  - HPCC HPL PTRANS benchmark
- Simple, general parallel Chapel code applies to any conformal matrix pair (dense, sparse, banded, distributed)

```
def transpose ( var A : [?A_domain] real,
                var C : [?C_domain] real )
      {
         forall (i,j) in C_domain do
            C[i,j] = beta*C[i,j] + A[j,i];
      }
```

Global Address Space with Affinity

- NOT a shared memory model
- Chapel domains encapsulate:
  - Index information: dimensions and included indices
  - Data distribution information: mapping to and layout on processors
- … enough information for compiler to address data, wherever it is and however it is laid out

Global Address Space with Affinity

2D-Block Cyclic declarations

```
const matrix_domain: domain (2)
                                    dmapped BlockCyclic ( lowIdx=(1,1),
                                                                          blk=(row_block_size, col_block_size) )
                      = [ 1..n_rows, 1..n_cols ],
           transpose_domain: domain (2)
                                           dmapped BlockCyclic  ( lowIdx=(1,1),
                                                                                  blk=(row_block_size, col_block_size) )
                       = [ 1..n_cols, 1..n_rows ];

    var A                                   : [matrix_domain   ] real,
          C                                 : [transpose_domain] real,
          C_save                            : [transpose_domain] real,
          C_plus_A_transpose : [transpose_domain] real;
```

Compiler now will compute all necessary addresses….

Exploiting Blocks and Locality

… beyond expectations for compiler technology – how can programmer help?

- Domains provide names for sets of indices
- Iterators provide names for sequences of sets of indices.

```
forall block in block_partitioning (C_domain) do
    forall (i,j) in block do
        C [i,j] = beta * C [i,j]  +  A [j,i];
```

- Outer forall iterates over all blocks in C; inner forall iterates over entries in a single block.
- In some computations we may need an “on clause” to tie data and computation together.

Iterators and Sequences of Index Sets

- An iterator is a single loop, each invocation delivering an index, a range of indices, tuples of indices or ranges, domains or more general objects.
- Messy index calculations isolated to a single location.

```
iter block_partitioning ( C_domain ) {

     const row_block_size = C_domain.blk(1),
                 col_block_size  = C_domain.blk(2);

     for (row_low, col_low) in C_domain
            by (row_block_size, col_block_size) do

       yield C_domain [ row_low .. #row_block_size,
                                   col_low  .. #col_block_size ];     }
```

Expressing Parallelism

- Data parallelism is expressed by *forall* and array operations.
- Chapel has multiple ways to describe task parallelism. We demonstrated two:
  1. *coforall* to spawn a single task per processor, enabling a standard SPMD implementation.
  2. asynchronous tasks with *sync* variables for a data-flow implementation.

- Dense linear algebra codes available at sourceforge.net.
  - PTRANS – point and blocked, in Chapel data parallel and SPMD form. For comparison, also in { Fortran77, Fortran 95 and C } + MPI, Co-Array Fortran, and UPC.
  - Dense Cholesky –
    - Point and outer product blocked using Chapel data parallel constructs
    - “Elemental” factorization in SPMD form
    - Data-flow blocked factorization

What Did We Learn?

Memory Addressing Model makes a huge difference

- Removes details and opportunities for error
- Changes what one needs to know.

For PTRANS benchmark:

- Chapel programmer's focus is completely on matrices.
- PGAS languages (one-sided messaging) requires details of block cyclic distribution.
- MPI (two-sided messaging) – synchronization requires processor-centric, not matrix-centric, view. Becomes an inverse communications problem:
  - In this case, an analytic solution exists. Not in HPCC benchmark.
  - Build a mapping table and invert it. Sufficiently complicated that HPCC benchmark has neither space nor time complexity correct.

What Did We Learn?

Size comparison of codes

all codes written by same programmer, equally well-commented, with error handling and task specialization

	C + MPI	Fortran + MPI	UPC	CAF	Point Chapel	Block Chapel	SPMD Chapel
Lines of code	1599 (+ header files)	1560	513	585	39	31	53
Tokens	5623	4663	1474	1400	181	241	444
(relative)	---	83%	26%	25%	3%	4%	8%

What Else Did We Learn?

Some features of Chapel are incomplete. We want / need

- Parallelism constructs for subsets / teams of processors
- More facilities for data replication and redundant computation
- More synchronization primitives
- Bulk communications primitives?

Chapel is a work in progress, funded only for a prototype compiler

- Performance is often lacking
- Some features were defined, but not yet implemented.

Graph Mini-application – SSCA 2

DARPA HPCS collection of “Synthetic Scalable Compact Applications”, #2 (http://www.highproductivity.org/SSCABmks.htm)

- Compute “(Approximate) Betweenness Centrality” for an arbitrary graph -- a measure of the relative importance of nodes in a network

- Similar to computations in sparse matrix reordering or partitioning.

Betweenness Centrality & Breadth-First Search

Betweenness Centrality -- in essence, a breadth-first search with memory

Two goals for Chapel code:

- Make code minimally dependent on details of graph data structures
- Test Chapel features for managing *sparse sets*

Graphs & Sparse Matrices: Generic Programming and Data Structures

- Typically:
  - Graph or sparse matrix representation is either implemented
    - in detail inline wherever needed.
    - via procedure calls.
  - Data distribution is hard-coded.

- Drawbacks:
  - Any hard-coded details are hard to maintain and change.
  - Procedure calls are inefficient for fine-scale operations.

- *Generic programming in Chapel successfully avoids both of these drawbacks*

Set Operations for Betweenness Centrality

- Build ***sets*** of nodes by
  - determining if a node is already in the set
  - If not, adding it to the set

  in parallel

- Iterate in parallel over the nodes of such a set

In most languages, must roll your own data structures. Parallel versions are complex.

Chapel domains include sparse and associative versions. Both natively support these set operations with parallel infrastructure.

Complete Betweenness Centrality Code

```
class Level_Set {
  type Sparse_Vertex_List;
  var Members   : Sparse_Vertex_List;
  var previous : Level_Set (Sparse_Vertex_List);}

def approximate_betweenness_centrality ( G, starting_vertices,
                                         out Between_Cent )
{ type Sparse_Vertex_List = sparse subdomain ( G.vertices );
  Between_Cent = 0;

  forall s in startin

    { var min_distance : [G.vertices] int      = -1;
      var path_count   : [G.vertices] int (64) = 0;
      var depend       : [G.vertices] real     = 0.0;

      var Active_Level = new Level_Set (Sparse_Vertex_List);
      var Next_Level   = new Level_Set (Sparse_Vertex_List);
      var current_distance : int = 0;
      min_distance (s)           = 0;

      Active_Level.Members.add ( s );
      Active_Level.previous = nil;
      Next_Level.previous    = Active_Level;
      Next_Level.Members.clear ();

      path_count (s)= 1;
```

```
while Active_Level.Members.numIndices > 0 do
  { current_distance += 1;
    forall  v in Active_Level.Members  do
      { forall (w, wt) in (G.Neighbors (v), G.edge_weight (v)) do
        { atomic
          if ( min_distance (w) < 0 && wt % 8 != 0 ) then
          { Next_Level.Members.add (w);
            min_distance (w) = current_distance; };};

        if ( min_distance (w) == current_distance ) then
                                           t (w) + path_count (v);

      Active_Level = Next_Level;
      Next_Level   = new Level_Set (Sparse_Vertex_List);
      Next_Level.Members.clear ();
      Next_Level.previous = Active_Level; };
  // end of forward pass

  var graph_diameter = current_distance - 1;
  Active_Level = Active_Level.previous;

  for current_distance in 2 .. graph_diameter by -1 do
    { Active_Level = Active_Level.previous;
      forall v in Active_Level.Members do
        { depend (v) = + reduce
                 [ (w, wt) in (G.Neighbors (v), G.edge_weight (v)) ]
                   if ( min_distance (w) == current_distance  &&
                          wt % 8 != 0 ) then
                     ( path_count (v):real / path_count (w) )
                      * ( 1 + depend (w) );
          atomic Between_Cent (v) += depend_v; }};
     // end of backward pass

    }; // outer embarassingly parallel forall
  return ( Between_Cent );}
```

```
Sparse_Vertex_List = sparse subdomain ( G.vertices )
```

```
forall  v in Active_Level.Members  do
```

```
if (
    {
```

```
  ( G.Neighbors (v),
    G.edge_weight (v) )
do
```

What Did We Learn?

- Chapel generic programming constructs freed us from almost all details of graph representation
  stay tuned for the next talk!
- Either sparse or associative domains provide all necessary set operations
- Fully functional Chapel parallel code resembles high-level “pseudo-code” in algorithmic papers
- Graph codes are dramatically shorter and simpler

Conclusions

- It works! Chapel really does make programming linear algebra algorithms in parallel much simpler.
- We need to test the language further, to ensure the final language design does what we need.
- This is a valuable effort – DARPA enabled us to start. The next step is funding beyond the prototype compiler.

The Really Cool Results are in the AMR Talks Coming Up Next

Thank You.

This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0001. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Defense Advanced Research Projects Agency

http://chapel.cray.com/

Questions?