Code Generation for Parallel Execution of a Class of Irregular Loops on Distributed Memory Systems

Mahesh Ravishankar\textsuperscript{1}, John Eisenlohr\textsuperscript{1}, Louis-Noël Pouchet\textsuperscript{1,3}, J. Ramanujam\textsuperscript{2}, Atanas Rountev\textsuperscript{1}, P. Sadayappan\textsuperscript{1}

\textsuperscript{1}Dept. of Computer Science & Engineering \hspace{1cm} \textsuperscript{2}School of EECS and Ctr. Comp. & Tech
\textsuperscript{3}Computer Science Department
\textsuperscript{1}The Ohio State University \hspace{1cm} \textsuperscript{2}Louisiana State University
\textsuperscript{3}University of California, Los Angeles
Motivation

• Converting sequential code to distributed memory code is challenging
  – Create parallel data structures
  – Manage communication
  – Difficult to debug

• Let the compiler do the heavy-lifting
  – Efficient code for affine computations
  – Many applications are “non-affine”

• Use a combination of static and run-time analysis
Irregular Computation: SpMV

\[
\begin{align*}
&\text{for}(i = 0; i < N ; i++) \\
&\quad \text{for}( j = 0; j < N; j++) \\
&\quad \quad y[i] += A[i][j] \times x[j]
\end{align*}
\]
Irregular Computations

- Data dependent control flow
- Data dependent array access pattern
- Multiple levels of indirection
- Target Loops that are parallel
  - Allows dependences due to commutative and associative reduction operations
  - Allow sparse reductions, e.g., $y[b[i]] +=$

```
for( i = 0 ; i < N ; i++)
    for( j = row[i]; j < row[i+1] ; j++)
        y[i] += A[j] * x[col[j]]
```
Generating distributed memory code

• Combined static/dynamic analysis
  – Static analysis for checking parallelism
  – Generate code which performs the run-time analysis, i.e. inspector code
  – Executor code performs the computation

• For efficient distributed memory execution
  – Inspector has to be parallel
  – Avoid complete replication of any data structure
  – Executor must preserve properties of original code that improve per node performance
Parallelization Strategy

- **Disjoint partitioning of iterations of partitionable loops**
  - no. of partitions = no. of processes
- **Partition the data**
  - Local arrays represent all original elements touched by iterations mapped to a process
- **Create auxiliary arrays on each process to recreate the control flow and data access pattern**
- **This functionality is provided by the inspector**
Auxiliary Arrays : Recreating the data access patterns

- Remember loop bounds of inner loops.
- Remember indices of arrays accessed.
- Modify these to point to local memory location.

```
loop_j = 0;
body_j = 0;
for( i = ...... ){
    lb[loop_j] = row[i];
    ub[loop_j] = row[i+1];
    for( j = row[i] ; j < row[i+1] ; j++ ){
        A_index[body_j] = j; //record index of A
        body_j++;
    }
    loop_j++;
}
```
Ghost Cells

- Same element might be touched by iterations on multiple processes
  - Every element has a unique owner
  - On other processes, treat them as ghost elements

```
col = 0 1 0 2 3 1 2 0 3 1 2 4 2 5

i = 0 1 2 3 4 5

Process 0: i = 0, 1, 2
0 1 0 2 3 1 2

Process 1: i = 3, 4, 5
0 3 1 2 4 2 5
```
Ghost Cells

- Ghosts cells within *read arrays* updated before loop is executed
- Ghosts cells within *updated arrays*
  - Initialized to 0 at the start of the loop
  - Contain partial contributions after loop execution
  - Communicated to owner for aggregation
- Enable execution of all iterations without communication
- **Number of ghosts cells represents the amount of communication**
Reducing the number of ghost cells

- Partitioning based on iteration-to-data affinity
  - Fewer ghosts => Less Communication
- Iteration-to-data affinity is represented by a hypergraph
Hypergraph Representation

- Hypergraph, $H = (V,E)$
  - Vertex $v$ represents an iteration of the partitionable loop
  - Hyper-edge $e$ represents a data element of the computation
  - Hyper-edge connects all vertices that touch the same data element

- Partitioning the hypergraph => Partitioning the computation
  - Load balance the iterations
  - Reduce the number of edges cut => reduction in ghost elements
Exploiting Contiguous Accesses

- Elements of A accessed in contiguous manner
- Local arrangement preserves contiguity
- Store only first element of each sequence
- Remaining elements obtained by an offset

A = [1, 2, 4]

On Process 1: i = 1

A_local = [0, 2, 5]

A_start = [0, 2, 5] + offset
Exploiting Contiguous Accesses

- What is the use of contiguity
  - Smaller auxiliary arrays => Smaller footprint
  - Helps with optimizations like prefetching, etc.
- Static analysis can reveal which expressions result in contiguous accesses
  - Elements accessed through such expressions remembered by the inspector
- Arranged in ascending order of original index
  - Maintains contiguity even in partially overlapping sequences
Parallel Inspector

• Inspector work-flow

  Building and partition the hypergraph

  Compute sizes of auxiliary arrays

  Populate and renumber values

• In parallel without complete replication of any of the original arrays
• Phase I – Analyze a block of iterations on each process
• Phases II & III – Analyze the iterations mapped to a process
• Challenge: Indirection Arrays are partitioned too
Related Work

• Joel Saltz group
  – CHOAS libraries : one-level of indirection
  – Compiler framework : Could not handle all sparse reductions
• Basumallik and Eigenmann (PPoPP 06)
  – Requires replication of shared data – higher foot-print
• LaMielle and Strout (CSU TR ’10, LCPC ‘12)
  – Sparse Polyhedral Framework
  – A more general approach, additional constraints needed to exploit contiguity, etc.
• more discussed in the paper
Benchmarks

• Evaluation of approach
  – Benchmarks: 183.equake from SPEC 2000, Conjugate Gradient Kernel, etc.
  – Production Application – Ocean, Land and Atmosphere Modeling (OLAM)
  – Compared against manual MPI implementations

• Contain a sequence of partitionable loops
  – Inspectors of all loops were merged

• Enclosed within an outer sequential loop
  – Inspector hoisted out for amortizing costs
Experimental Setup

• Code generation algorithm implemented within the ROSE compiler framework
• Partitioners
  – PaToH – Sequential hypergraph partitioner
  – ParMetis – Parallel Graph Partitioner
    • Convert the hypergraph to equivalent graph
• Cluster with Intel Xeon E5460 processor – 4 cores per node
• Communication implemented using ARMCI one sided communication operations, GA – 5.1
- Sparse Matrix: 220,542 rows; 9,895,422 non-zeros
- Lowest Execution Times: PaToH and ParMetis
  - 478.15s => 8.3s, with inspector costs => 13.31s
- PetSc implementation: CG solver, no pre-conditioning
  - Arrays rows block-partitioned across the solver
• Unstructured grid: Tetrahedral elements
• Performance comparable to manual implementation
  – Minimum Execution time: 23.23s => 2.1s
• Negligible inspector times with ParMetis
  – At 64 processes, 0.23s
• FORTRAN 90 application
  – complex and inter-procedural
• Outer time loop executes hundreds of thousands of iterations
  – Here only 30000 time steps
• Generated code scales linearly
  – manual version scales super-linearly due to smaller footprint attained due to domain specific knowledge
Conclusion

- Framework to automatically generate distributed memory code
  - Partitioning the computation based on iteration-to-data affinity
  - Parallel inspector without complete replication of any original data structures
  - Generated code maintains contiguity of accesses to data
    - Enable subsequent optimizations such as prefetching
- Performance is comparable to manual MPI implementation
  - when inspector cost can be effectively amortized
Thank You

• Acknowledgements
  – Prof. Umit Catalyurek, Prof. Gagan Agrawal, Prof. Robert Walko, and the reviewers

• More detailed technical report available at www.cse.ohio-state.edu/~ravishan/

Questions?