Data-parallel Abstractions for Irregular Applications

Keshav Pingali
University of Texas, Austin

Joint work with Milind Kulkarni, Martin Burtscher, Ganesh Ramanarayanan, Bruce Walter, Kavita Bala, Paul Chew

Background

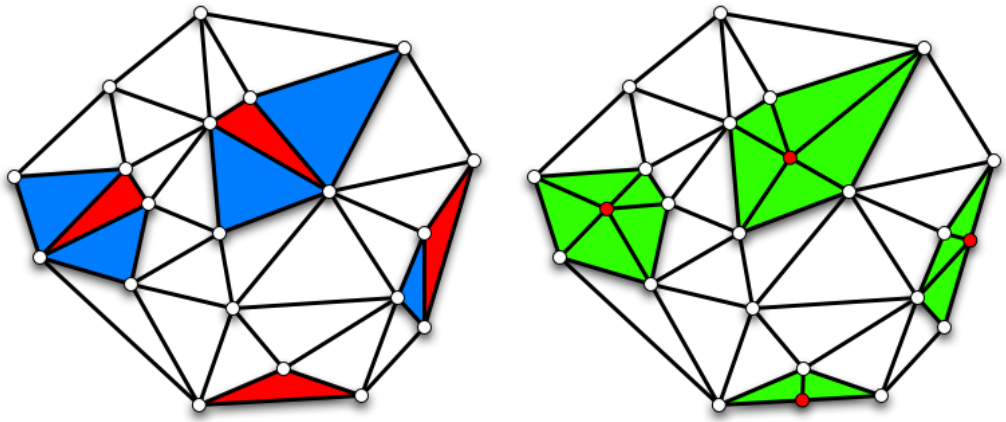
- Regular programs
  - Structured data such as arrays and relations
  - Simple data access patterns such as fixed-stride accesses
  - Application domains: databases, computational science
  - We understand parallelism and locality in these domains
  - Lots of language support, compiler techniques and tools
- Irregular programs
  - Unstructured data such as lists, trees, graphs built from pointers
  - Complex data access patterns
  - Programs in most application domains are irregular
  - We understand very little about parallelism and locality in irregular programs, let alone how to provide language constructs and system support for such programs
  - My belief:
    - program = algorithm + data structure
    - To make progress, we must study algorithms, not programs

Main points

- Irregular programs have data-parallelism
  - Work-list based iterative algorithms over irregular data structures
- Optimistic parallelization is essential for such apps
  - Parallelism may be inherently data-dependent
  - Pointer/shape analysis cannot work for these apps
- Expressing and exploiting irregular data-parallelism
  - Algorithms:
    - expressed using iterators over unordered and ordered sets
  - Data structures:
    - abstractions provided by object-oriented programming are critical
    - high-level semantic information is important
  - Generalization of approach for regular data-parallelism
- Galois system: implementation of these ideas
  - Also includes data partitioning and support for scheduling iterator iterations (PLDI 2007, ASPLOS 2008, PLDI 2008)

Parallelism case studies: two irregular programs

Delaunay Mesh Refinement



- Initial mesh has bad triangles
- Iterative refinement procedure:

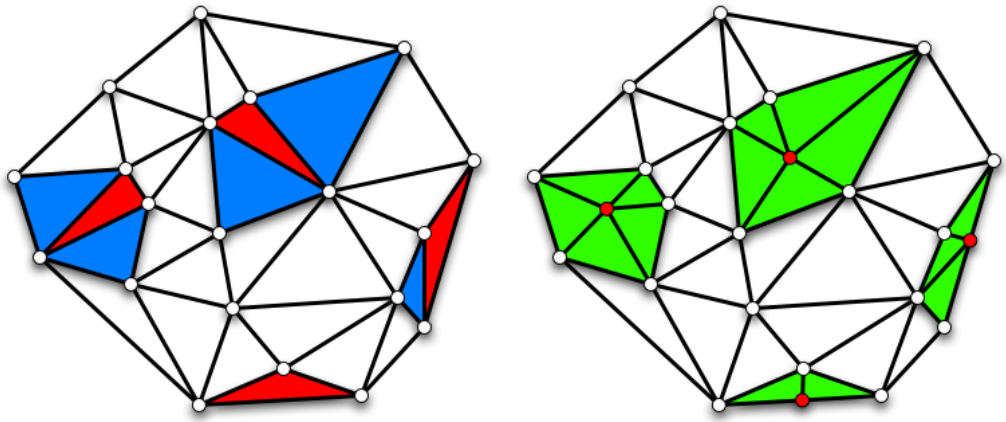
```
while there are bad triangles do {
        Pick a bad triangle;
        Find its cavity;
        Retriangulate cavity; // may create new bad triangles
        }
```

- Order in which bad triangles should be refined:
  - Final mesh depends on order in which bad triangles are processed
  - But all bad triangles will be eliminated ultimately regardless of order

Sequential Algorithm

```
Mesh m = /* read in mesh */
WorkList wl;
wl.add(mesh.badTriangles());
while (true) {
        if ( wl.empty() ) break;
        Element e = wl.get();
        if (e no longer in mesh) continue;
        Cavity c = new Cavity(e);//determine new cavity
        c.expand();
        c.retriangulate();//re-triangulate region
        m.update(c);//update mesh
        wl.add(c.badTriangles());
}
```

Parallelization opportunities

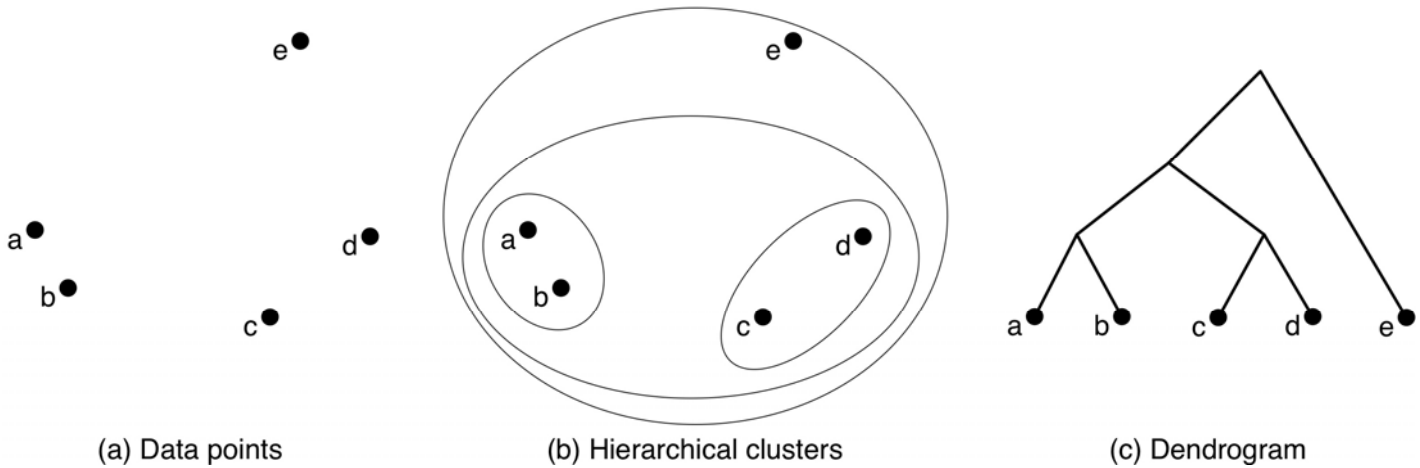


- Triangles with non-overlapping cavities can be processed concurrently
  - if cavities of two triangles overlap, they must be done serially
- Any compile-time parallelization scheme must be conservative and assume that dependences might exist

➔ Parallel execution requires runtime dependence checking

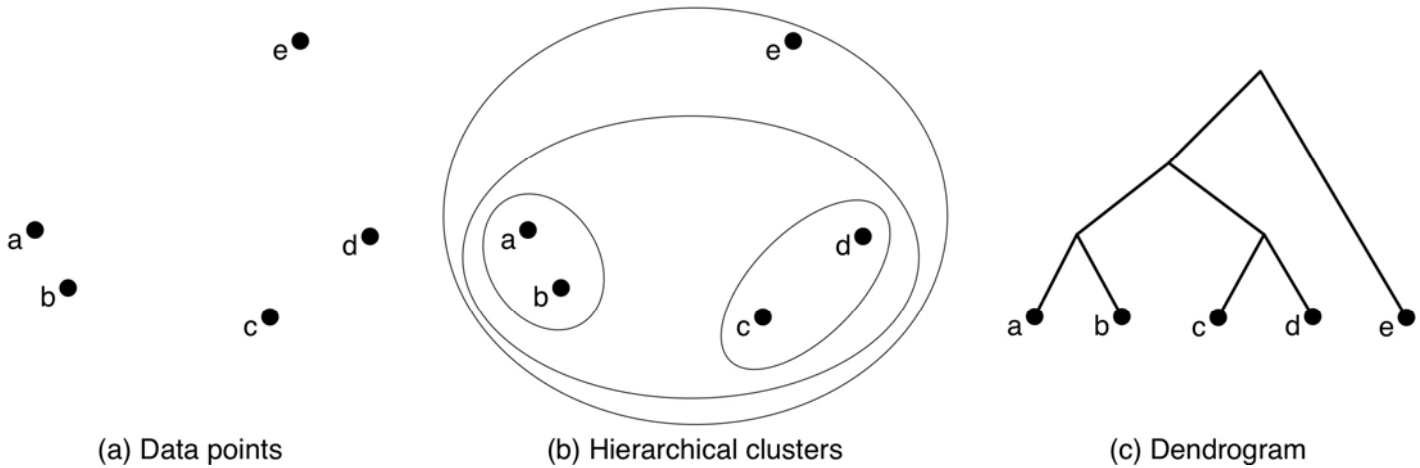
  - Property of algorithm, not program

Agglomerative Clustering



- Input:
  - Set of data points
  - Measure of “distance” (similarity) between them
- Output: dendrogram
  - Tree that exposes similarity hierarchy
- Applications:
  - Data mining
  - Graphics: lightcuts for rendering with large numbers of light sources

Clustering algorithm



- Sequential algorithm: iterative
  - Find two closest points in data set
  - Cluster them in dendrogram
  - Replace pair in data set with a “supernode” that represents pair
    - Placement of supernode: use heuristics like center of mass
  - Repeat until there is only one point left

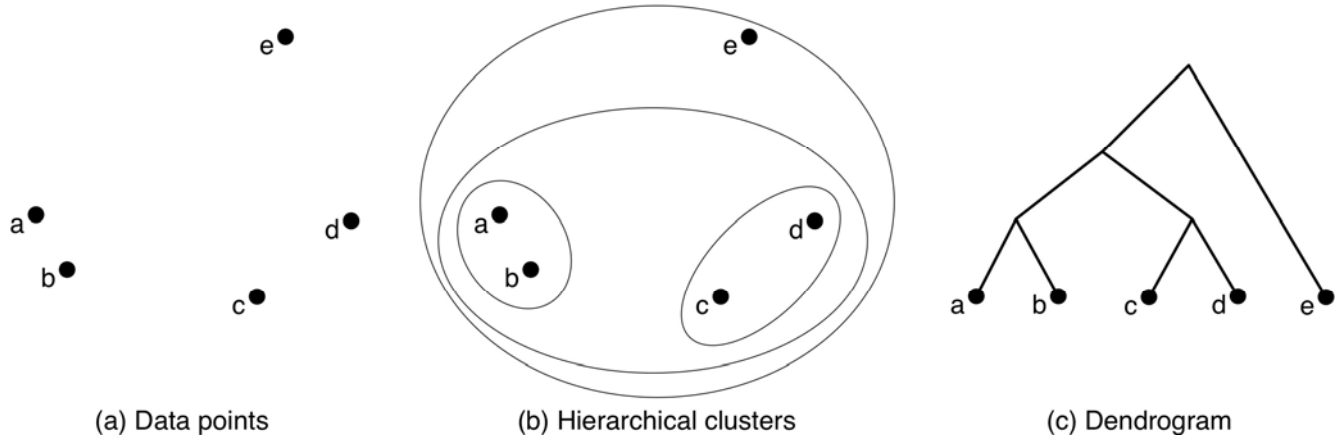
Key Data Structures

- Priority queue:
  - Elements are pairs <p,n> where
    - p is point in data set
    - n is its nearest neighbor
  - Ordered by increasing distance
- kdTree:
  - Answers queries for nearest neighbor of a point
  - Convention: if there is only one point, nearest neighbor is point at infinity (ptAtInfinity)
  - Similar to a binary search tree but in higher dimensions

Clustering algorithm: implementation

```
kdTree := new KDTree(points);
pq := new PriorityQueue();
for each p in points (pq.add(<p,kdTree.nearest(p)>));

while (true) do {
   if (pq.size() == 0) break;
   pair <p,n> := pq.get(); //get closest pair
   …….
   Cluster c := new Cluster(p,n); //create supernode
   dendrogram.add(c);
   kdTree.remove(p); //update kdTree
   kdTree.remove(n);
   kdTree.add(c);
   Point m := kdTree.nearest(c); //update priority queue
   ……. pq.add(<c,m>);
}
```

Parallelization Opportunities



- Natural unit of work: processing of a pair in PQ
- Algorithm appears to be sequential
  - pair enqueued in one iteration into PQ may be the pair dequeued in next iteration
- However, in example, <a,b> and <c,d> can be clustered in parallel
- If dendrogram is bushy tree, lots of opportunities for parallelism
  - but parallelism is very data-dependent: compile-time parallelization cannot work

Take-away lessons

- Irregular programs have data-parallelism
  - Data-parallelism has been studied in the context of arrays
  - For unstructured data, data-parallelism arises from work-lists of various kinds
    - Delaunay mesh refinement: list of bad triangles
    - Agglomerative clustering: priority queue of pairs of points
    - Maxflow algorithms:list of active nodes
      - Boykov-Kolmogorov algorithm for image segmentation
      - Preflow-push algorithm
    - Approximate SAT solvers
    - …….
- Data-parallelism in irregular programs is obscured within while loops, exit conditions, etc.
  - Need transparent syntax similar to FOR loops for structured data-parallelism

Take-away lessons (contd.)

- Parallelism may depend on “data values”
  - whether or not two potential data-parallel computations conflict may depend on input data
    - (e.g.) Delaunay mesh generation: depends on shape of mesh
- Optimistic parallelization is necessary in general
  - Compile-time approaches using points-to analysis or shape analysis may be adequate for some cases
  - In general, runtime conflict-checking is needed
- Handling of conflicts depends on the application
  - Delaunay mesh generation: roll back all but one conflicting computation
  - Agglomerative clustering: must respect priority queue order

Galois programming model and implementation

Beliefs underlying Galois system

- Optimistic parallelism is the only general approach to parallelizing irregular apps
  - Static analysis can be used to optimize optimistic execution
- Concurrency should be packaged within syntactic constructs that are natural for application programmers and obvious to compilers and runtime systems
  - Libraries/runtime system should manage concurrency (cf. SQL)
  - Application code should be sequential
- Crucial to exploit abstractions provided by object-oriented languages
  - in particular, distinction between abstract data type and its implementation type
- Concurrent access to shared mutable objects is essential

Components of Galois approach

1) Two syntactic constructs for packaging optimistic parallelism as iteration over sets
2) Assertions about methods in class libraries
3) Runtime system for detecting and recovering from potentially unsafe accesses by optimistic computations

(1) Concurrency constructs: two set iterators

- *for each e in Set S do B(e)*
  - evaluate block B(e) for each element in set S
  - sequential implementation
    - set elements are unordered, so no a priori order on iterations
    - there may be dependences between iterations
  - set S may get new elements during execution
- *for each e in PoSet S do B(e)*
  - evaluate block B(e) for each element in set S
  - sequential implementation
    - perform iterations in order specified by poSet
    - there may be dependences between iterations
  - set S may get new elements during execution

Galois version of mesh refinement

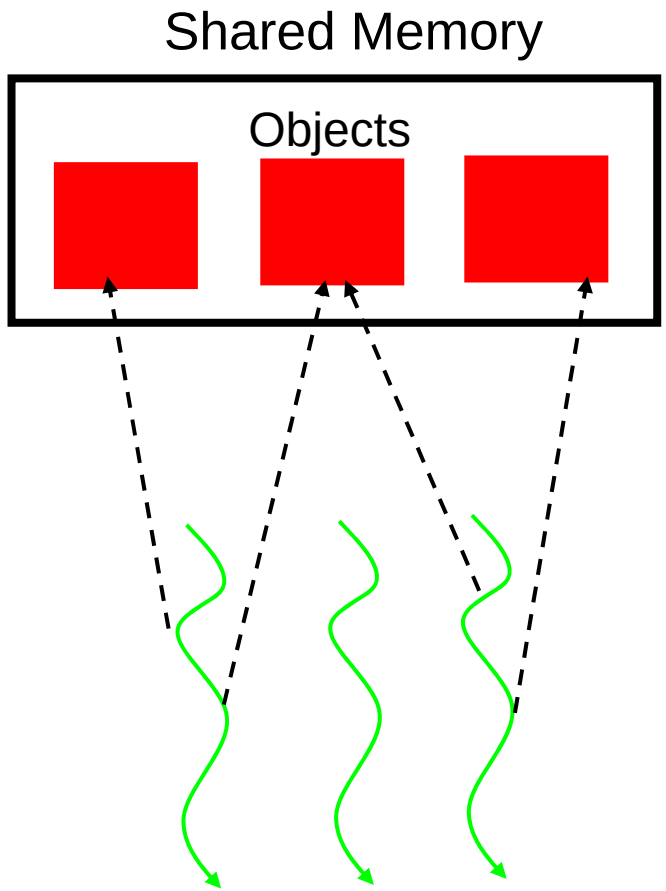
```
Mesh m = /* read in mesh */
Set wl;
wl.add(mesh.badTriangles()); // non-deterministic order

for each e in Set wl do {                    //unordered iterator
        if (e no longer in mesh) continue;
        Cavity c = new Cavity(e);            //determine new cavity
        c.expand();                          //determine affected triangles
        c.retriangulate();                   //re-triangulate region
        m.update(c);                         //update mesh
        wl.add(c.badTriangles());            //add new bad triangles to workset
}
```

Observations

- Application program has a well-defined sequential semantics
  - No notion of threads/locks/critical sections etc.
- Set iterators
  - SETL language was probably first to introduce set iterators
  - However, SETL set iterators did not permit the sets being iterated on to grow during execution, which is important for our applications
- Generalization of regular program constructs
  - FORTRAN-style DO loops are iterators over integer sets
    - DO-ALL loop: special case of unordered set iterator
    - Non-DO-ALL loop: special case of ordered set iterator

Parallel execution model

- Object-based shared-memory model
- Computation performed by some number of threads
- Threads can have their own local memory
- Threads must invoke methods to access internal state of objects
  - mesh refinement:shared objects are
    - worklist
    - Mesh
  - agglomerative clustering
    - priority queue
    - kdTree
    - dendrogram

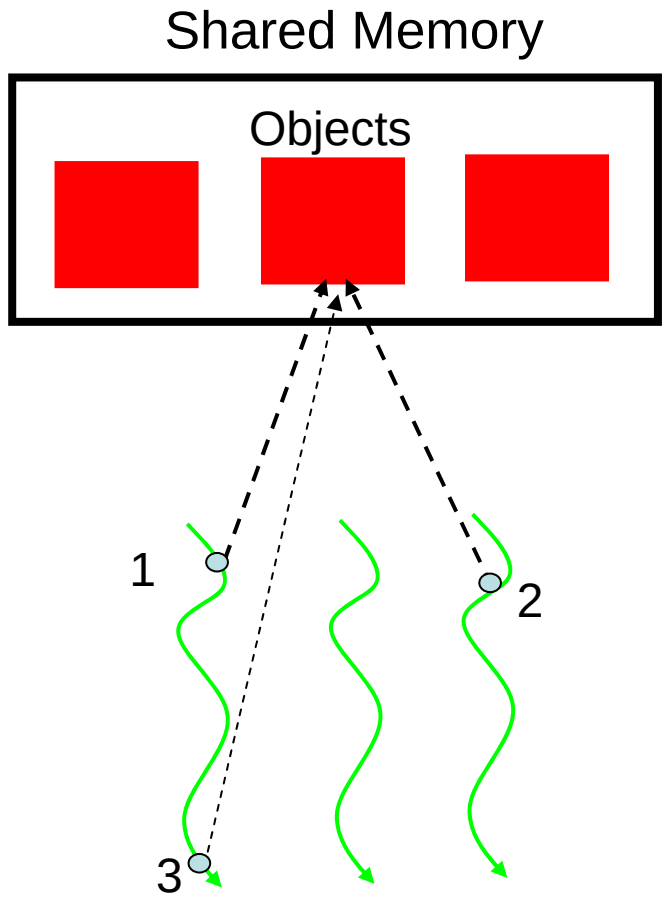


Parallel execution of iterators

- Master thread and some number of worker threads
  - master thread begins execution of program and executes code between iterators
  - when it encounters iterator, worker threads help by executing some iterations concurrently with master
  - threads synchronize by barrier synchronization at end of iterator
- Key technical problem
  - Parallel execution must respect sequential semantics of application program
    - result of parallel execution must appear as though iterations were performed in some interleaved order
    - for poSet iterator, this order must correspond to poSet order
  - Non-trivial problem
    - each iteration may access mutable shared objects

(2) Class libraries

- Complexity of concurrency control is hidden within library
- Mutual exclusion:
  - To invoke method, thread acquires lock on object, performs method and releases lock
- Serializability:
  - Two-phase locking usually limits concurrency
  - Another approach: exploit commutativity of method invocations
  - Semantic commutativity, not representational commutativity, as specified by user
- Back-off:
  - Each class method must have an inverse that undoes the effect of that method
  - Semantic inverse, not representational inverse



Example: set

```
Class SetInterface {
  void add (Element x);
     [conflicts]
       - add(x)
       - remove(x)
       - contains?(x)
       - get() :x
      [inverse] remove(x)
    void remove(Element x);
       [conflicts]
        - add(x)
        - remove(x)
        - contains?(x)
        - get(): x
       [inverse] add(x)
       ……
}
```

(3) Runtime system

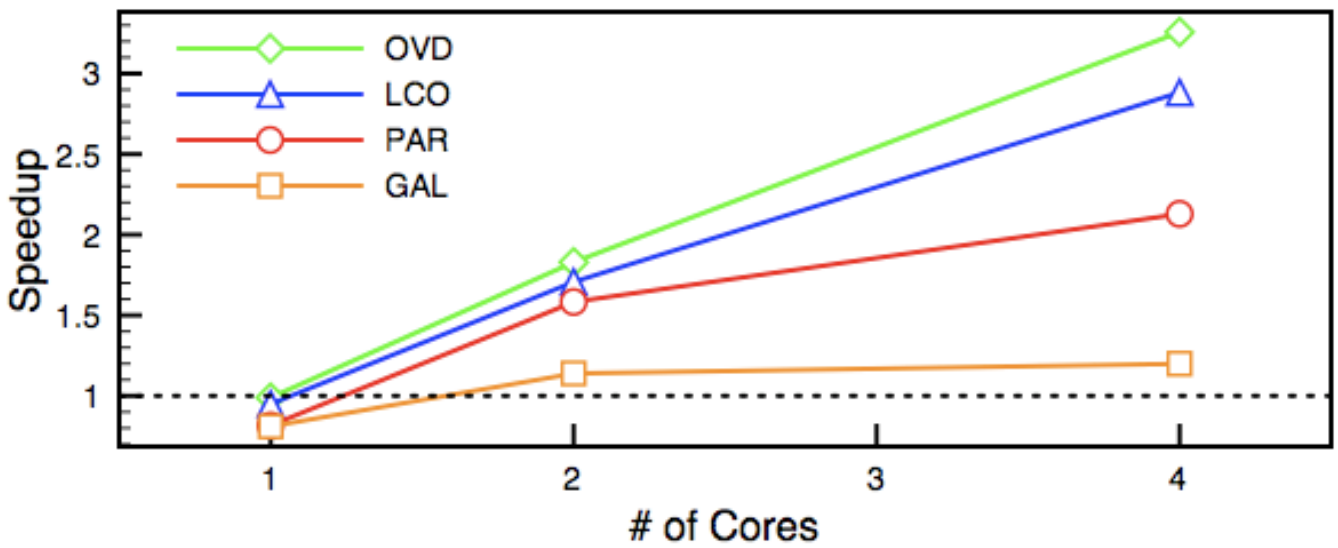
- Role is similar to that of reorder buffer in out-of-order processors
  - Maintain record for each iteration
  - Track method invocations made by that iteration
  - Take action to roll iteration back on conflicts
  - Commit iteration when it reaches head of reorder queue
- Objects log method invocations made by ongoing iterations
  - When iteration commits, its method invocations are removed from logs of all objects

Two extensions

- Data structure partitioning
  - Improve locality
  - Reduce speculation conflicts
  - Reduce overhead of commutativity checks:
    replace commutativity checks with two-phase locking on partitions
  - To appear in ASPLOS'08
- Scheduling framework
  - Specify application-specific schedule
  - Generalizes Open-MP style schedules (static/dynamic/guided self-scheduling)
  - Under submission

Results for Mesh Refinement

- 4 processor Itanium-2
- Versions:
  - GAL: using stack as worklist
  - PAR: partitioned mesh
  - LCO: locks on partitions
  - OVD: over-decomposed version



Conclusions

Unified approach to parallelism
in regular and irregular programs

- Data parallelism:
  - regular programs: matrix computations
  - Irregular programs: worklist-based iterative algorithms
- Syntactic constructs:
  - regular programs: DO loops and DO-ALL loops
  - irregular programs: Ordered and unordered set iterators
- Parallelism in data structure operations
  - Regular programs: reads and writes to different matrix locations
  - Irregular programs: commutativity of method invocations
- Loop-level parallelism:
  - DO-ALL loops: no dependences between iterations
  - Irregular programs: may or may not be dependences between iterations
- Data structure partitioning
  - Regular programs: arrays
  - Irregular programs: graphs, trees, lists, ….
- Iteration scheduling:
  - Generalizes OpenMP schedules