Runtime and Compilation Methods for Irregular Computations on Distributed Memory Parallel Machines
Outline

Target Problems

Parallelizing Irregular Problems

Strategy: Runtime, Language and Compiler Support
Target Problems

Parallelizable Loops with Communication Patterns Known Before Executing the Loops

Data Access Pattern May or May Not Change between Iterations or Time steps

Computational Load May Change between Time steps

Examples:

- Explicit Multigrid Unstructured CFD Solvers
- Molecular Dynamics Codes (CHARMM, AMBER, GROMOS etc)
- Time Dependent Flame Modeling Codes
Sweep Over an Unstructured Mesh - an Irregular Loop

do i = 1, n_step ! Outer Loop L1
...
do i=1, nedge ! Inner Loop L2
  y(edge1(i)) = y(edge1(i)) + x(edge1(i))
  + x(edge2(i))
y(edge2(i)) = y(edge2(i)) + x(edge1(i))
  + x(edge2(i))
end do
end do
draw

- Values of edge1 and edge2 are Known Only at Runtime
Parallelization

- Data Arrays and Loop Iterations Distributed
- Fetch On-Demand
  - Another Strategy
    - Inspector/Executor Strategy
      * At Runtime Pre-process Indirection Arrays
      * Generate Communication Calls
      * Execute Loop
Issues to Be Addressed

Data Partitioning
Work Partitioning – Load Balancing
Communication Volume
Communication Startup Costs

- Techniques for Runtime and Compiler Support must
  Address These Issues in a Systematic and
  Comprehensive Manner
A Methodology for Parallelizing Irregular Problems

Phase A
Compute Maparray Values

Phase B
Remap Data Arrays

Phase C
Generate Iteration Graph
Partition Iteration Graph

Phase D
Remap Loop Iterations

Phase E
Pre-process Loops

Phase F
Execute Loops
Data Management

- A Global Name Space Program Translated to Local Name Space Program + MP
- Requires Runtime Support
  - Buffer Management
  - Index Translation
Data Management

Partitioned global reference list

localized global reference list

local buffer references

off-processor references

local buffer

data array

local data

buffer

off-processor data
E.G., CHAOS Runtime Support

(Ponnusamy, Saltz, Choudhary)

CHAOS Functionality:

- Provides Support for Irregular Distributions
- Distributes Loop Iterations
- Generates Send/Receive Communication Calls
- Eliminates Duplicate Off-processor Data Transmission
- "Vectorizes" Communication
- Translates Global to Local References
Runtime Support Cont.

(Existing Maryland/Syracuse Runtime Support)

Distributed Array Mapping

Standardized distributed representation of connectivity graph, computational load and geometric information - (GeoCoL Graph)

runtime support used to generate Data Graph

partitioners assign array elements to processors based on Data Graph

Array Remapping

remap between irregularly distributed arrays and between regularly and irregularly distributed arrays
Index Array Translation, Buffer Allocation
  indirection pattern analyzed
  • home processors found for each array index
  • communication pattern extracted
  • off-processor data copies assigned to buffer locations

Partition loop iterations or indirection array
  given array partition find judicious partition of loop iterations, or
  given array partition, partition indirection array to satisfy "owner computes"
Node Code with CHAOS Procedures

Create the required schedules (Inspector)
...
S1 call gather(x(begin_buffer), x, schedule)
S2 do i=1, n_local_edge
S3 y(local_edge1(i)) = y(local_edge1(i)) ... +
   x(local_edge1(i)) ... + x(local_edge2(i))
S4 y(local_edge2(i)) = y(local_edge2(i)) ... +
   x(local_edge1(i)) ... + x(local_edge2(i))
S5 end do
S6 call scatter_add(y(begin_buffer), y, schedule)
Types of Partitioners

Based on Proximity in Physical Space (i.e. on array values)
- Recursive Coordinate Bisection
- Recursive Bisection Based on Moments of Inertia

Based on Connectivity Structure (i.e. on Data Access Patterns)
- K-L Methods, Eigenvalue Graph Partitioning,
- Simulated Annealing, Neural Nets, Genetic Algorithms

Based on Computational Load Associated With Each Array Element
Irregular Distributions

Not in HPF-1 but covered in Fortran D, Vienna Fortran

Arrays may be irregularly distributed among processors, e.g. array elements embedded into processors based on partitioner results

Distributions can be specified using a map array - if map_array(i) = p, then array index i is mapped to processor p

Vienna Fortran provides interface to functions which return user defined distributions

Distributions can also be specified by symbolically
designating how a partition is to be generated
Simple Example of Fortran D Declarations

REAL x(N), y(N)
INTEGER map(N)
C$ DECOMPOSITION irreg(N), reg(N)
C$ ALIGN map with reg
C$ DISTRIBUTE reg(block)

... set values of map array using some mapping method...

C$ DISTRIBUTE irreg(map)
C$ ALIGN x, y with irreg
Vienna Fortran FORALL

Representation of reductions inside foralls
Declares independence of loop iterations (except for any reductions occurring in loop)
Allows user to partition loop iterations between processors

FORALL I=1,N ON OWNER(IA(I))
  DELTAX = F(X(IA(I)),X(IB(I)))
  REDUCE(ADD,Y(IA(I)),DELTAX)
  REDUCE(ADD,Y(IB(I)),-DELTAX)

END FORALL
Implicitly Specified Mappings

In current Fortran-D the User ExplicitlySpecifiesIrregular Array Distribution Using Map Arrays

Another Strategy is to Implicitly Specify Mappings

User Provides Symbolic Information

Compiler Generates Code To:

- Generate “GeoCoL” Data Structure
- Pass “GeoCoL” Data Structure to a Mapper
- Mapper Returns New Array Distribution
Compiler Embeddedable Partitioners

- Partitioning Strategy Can Be Specified By Directive

  User Chooses Among Library of Partitioning Methods
  User Also Chooses Criteria on Which Partitioning is to Be Based
  Extended the Fortran 90D Compiler to Embed Many Kinds of Partitioners
<table>
<thead>
<tr>
<th>Partitioner</th>
<th>Geometry Information</th>
<th>Connectivity Information</th>
<th>Vertex Weight</th>
<th>Edge Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spectral Bisection</td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Coordinate Bisection</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Hierarchical Subbox Decomposition</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Simulated Annealing</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Neural Network</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Genetic Algorithms</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Inertial Bisection</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Kernighan – Lin</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
GeoCoL Data Structure

- GeoCoL Data Structure Consists of
  - Geometry Information
  - Connectivity Information
  - Computational Load Information
Coupling Data Mappers to Compilers  

Language Extensions
- To Generate GeoCoL Graph
- To Invoke Partitioners

Compiler Generates code To:
- Generate GeoCoL Data Structure
- Redistribute Arrays
Language Extensions

REAL x(N), y(N)
INTEGER map(N)
C$ DECOMPOSITION irreg(N)
C$ CONSTRUCT G (N, LINK(E, edge1, edge2),
LOAD(wt), GEOMETRY(2, x, y))

C$ SET mydist BY PARTITIONING G USING
my_partitioner

C$ REDISTRIBUTE irreg(mydist)
Loop Iteration Mapping

- Each Iteration References Distributed Array Elements
- Owner Computes Rule:
  * Assign Iteration to Left Hand Side Owner
  * Cumbersome When There Are Multiple LHS Indirections in the Loop
- Almost Owner Computes Rule
  * Identify the Processors that Own Array References
  * Assign Iteration to the Processor that Owns Maximum number of Array References in that Iteration
Array and Loop Iterations Remapping

- Derive Communication Pattern (schedule) to Move Data From Old to New Mapping
- CHAOS Software Used to Schedule Communication and Transport Data
  * Communication is Vectorized
  * Schedules are Stored and Can Be Reused for Multiple Data Arrays
Communication Schedule Reuse in Compiler

- Pre-processing is Expensive
- Can Be Reused When all of:
  - Distribution of Data Arrays has not Changed
  - Indirection Arrays Have Not Been Modified
  - Loop Bounds Remain Unchanged
Schedule Reuse - Implementation.

- Strategy:
  - A global Time Stamp
  - A local Time Stamp
  - Update the Global Time Stamp
    Whenever an Array is Modified
    Whenever an Array Distribution Changes
  - Reuse Allowed If Global Time Stamp Matches Local Time Stamp
Schedule Reuse - Contd.

C Outer loop – **Without Schedule Reuse**
S1  Compute Schedules
S2  Execute Inner loop

C Outer loop – **With Schedule Reuse**
K1  Check If New Schedules Needed
    If needed
K2  Compute and Store New schedules
    Else
K3  Retrieve Old schedules
    Endif
K4  Execute Inner Loop
Fortran 90D Code

real*8 x(nnode), y(nnode)
integer edge1, edge2
S1 decomposition reg(nnode), reg2(nedge)
S2 distribute reg(BLOCK), reg2(BLOCK)
S3 align x, y with reg
S4 align edge1, edge2 with reg2

....
call read_data(edge1, edge2, ...)
S5 construct G (nnode, link(nedge, edge1, edge2))
S6 set distfmt by partitioning G using RSB
S7 redistribute reg(distfmt)
C Loop over edges involving x, y
L1 forall(i=1:nedge)
  S8 y(edge1(i)) = y(edge1(i)) + ... x(edge2(i))
  S9 y(edge2(i)) = y(edge2(i)) + ... x(edge1(i))
  end forall
... 
S3 CALL update_dad(1, BLOCK, nnode, nmod) 
S4 CALL update_dad(1, BLOCK, nedge, nmod) 
    CALL read_mesh (edge1, edge2 ...) 
C$ CONSTRUCT G (nnode, LINK (nedge, edge1, 
    edge2)) 
S5 hashptr = init_rdg_hash_table (nedge) 
S6 CALL eliminate_dup_edges (hashptr, edge1,edge2, 1, 
    nedge) 
C$ SET distfmt BY PARTITIONING G USING RSB 
S7 CALL generate_rdg (hashptr, nnode, csr_ptr, csr_col, 
    ncol)
S8 CALL RSB (nnode, csr_ptr, csr_col, ncol, ntable)
C$ REDISTRIBUTE reg (distfmt)
S9 CALL remap (ntable, rsched, 1, nnode, local_nnode)
S10 CALL dgather (y (1), y (1), rsched)
S11 CALL dgather (x (1), x (1), rsched)
S12 CALL update_mod (BLOCK, IRREG, nnode, nmod)
Compiler Generated Code Contd.

C$ Parallelized irregular loop - inspector and executor with schedule saving
S13 Check if new inspector needed
   If needed
S14   call CHAOS procedure to Partition loop iterations
S15   call CHAOS procedure to Compute new schedule
S16   Store new schedule and loop bound information
   Else
S17   Retrieve previous schedule and loop information
   Endif
S18   call CHAOS procedure to Gather off-processor data
S19 Execute loop
S20 call CHAOS procedure to Scatter off-processor data
...
Performance Results on Intel iPSC/860

Performance Results From Templates Written with Partitioner Extensions and Compiled With Fortran 90D Compiler

Template from Unstructured Mesh Euler Solver:
  • Data Access Pattern is Irregular But Static
  • Indirect Distributed Array References
    (e.g. \( y(ia(i)) = y(ia(i)) + x(ib(i)) \))

Template from Molecular Dynamics (CHARMM)
  • Access Pattern Changes Every Few Timesteps

Template from Hypersonic Combustion Code:
  • Computational Load Changes Every Timestep
## Performance of Schedule Reuse - Intel iPSC/860

<table>
<thead>
<tr>
<th></th>
<th>10K Mesh</th>
<th>53k Mesh</th>
<th>648 Atoms</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Time in Secs)</td>
<td>Processes</td>
<td>Processes</td>
<td>Processes</td>
</tr>
<tr>
<td>8</td>
<td>16</td>
<td>32</td>
<td>64</td>
</tr>
<tr>
<td>No Schedule Reuse</td>
<td>161</td>
<td>94</td>
<td>301</td>
</tr>
<tr>
<td>Schedule Reuse</td>
<td>10.1</td>
<td>8.4</td>
<td>19.8</td>
</tr>
</tbody>
</table>

- 100 Timesteps
## Performance

Unstructured 53K Mesh Template - 32 Processors

<table>
<thead>
<tr>
<th>(Time in Secs)</th>
<th>Recursive Coordinate</th>
<th>Bisection</th>
<th>Block</th>
<th>Partition</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Hand Coded</td>
<td>Compiler: No Schedule Reuse</td>
<td></td>
<td>Hand Coded</td>
</tr>
<tr>
<td>Partitioner</td>
<td>1.3</td>
<td>1.3</td>
<td>1.3</td>
<td>0.0</td>
</tr>
<tr>
<td>Inspector, remap</td>
<td>3.3</td>
<td>286</td>
<td>3.4</td>
<td>3.2</td>
</tr>
<tr>
<td>Executor</td>
<td>13.9</td>
<td>13.9</td>
<td>15.1</td>
<td>36.6</td>
</tr>
<tr>
<td>Total</td>
<td>18.5</td>
<td>301</td>
<td>19.8</td>
<td>39.8</td>
</tr>
</tbody>
</table>
Computational Loop Structure of CHARMM

L1: do i = 1, NATOM
    L2: do index = 1, INB(i)
        j = Partners(i, index)
        Calculate dF (x, y and z components).
        Subtract dF from $F_j$.
        Add dF to $F_i$
    end do
end do

• Partner List Modified Every Few Timesteps
Partitioning Using BLOCK
Using a Partitioner
Adaptive CHARMM

<table>
<thead>
<tr>
<th>(in sec.)</th>
<th>Proc</th>
<th>Partition</th>
<th>Remap</th>
<th>Inspector</th>
<th>Executor</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hand Coded</td>
<td>32</td>
<td>3.2</td>
<td>8.2</td>
<td>2.8</td>
<td>84.6</td>
<td>98.8</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>4.2</td>
<td>6.7</td>
<td>2.0</td>
<td>62.9</td>
<td>75.8</td>
</tr>
<tr>
<td>Compiler</td>
<td>32</td>
<td>3.3</td>
<td>8.7</td>
<td>3.1</td>
<td>85.0</td>
<td>100.1</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>4.3</td>
<td>7.1</td>
<td>2.2</td>
<td>63.6</td>
<td>77.2</td>
</tr>
</tbody>
</table>

- 14K Atoms
- 6 Million Edges
- 100 Timesteps
- Partner List Regenerated Every 10 Timesteps
Distributed Memory Irregular Problem Compilation

Some Researchers

- Berkeley
  Kathy Yelick
- Cornell
  Craig Chase, Anthony Reeves
- Drexel
  Robert Weaver
- ICASE
  Piyush Mehrotra
- Los Alamos
  
  Dan Quinlan

- University of Maryland
  
  Gagan Agrawal, Raja Das, Shin Hwang, Bongki Moon, Ravi Ponnusamy, Joel Saltz, Shamik Sharma, Alan Sussman, Mustafa Uysal

- Rutgers
  
  Tao Yang, Apostolos Gerasoulis

- Rice

  Reinhard von Hanxleden, Seema Hiranandani, Ken Kennedy, Chuck Koelbel
• Stanford
  Monica Lam, Steve Lundstrom
• Syracuse
  Alok Choudhary, Geoffrey Fox, Ravi Ponnusamy, Sanjay Ranka
• Yale
  Harry Berryman, Marina Chen
Initial Distribution
After Remapping