Locality Optimizations for Scientific Applications

Chau-Wen Tseng
(with Hwansoo Han & Gabriel Rivera)
Department of Computer Science
University of Maryland, College Park
What Is Locality?

- **Locality**
  - Multiple references to same / nearby locations

- **Types of locality**
  - **Temporal** (reuse data)
  - **Spatial** (reuse nearby data)
Importance of Locality

- **Locality is key to high performance**
  - Computers are designed to reward locality
  - Processors are much faster than memory
  - Caches hide speed difference

- **Growing processor-memory gap**
  - Differences in speed are increasing
  - Locality becomes more important
Processor vs. Memory Speed (Latency)

Year


Speed (Mhz)

3500 3000 2500 2000 1500 1000 500 0

Processor Clock

Memory Bus Clock

x86
x486
x486 DX2
x486 DX4
Pentium
Pentium Pro
Pentium II
Pentium III
P IV
P II MMX
EDO DRAM (300 ns)
SDRAM (200 ns)
DDR-DRAM (200 ns)
FPM DRAM (420 ns)
Processor vs. Memory Speed (log scale)

- x386
- x486
- Pentium
- x486 DX2
- x486 DX4
- Pentium Pro
- Pentium II
- P II MMX
- P III
- P 4
- DDR-DRAM

Year:
- 1988
- 1989
- 1990
- 1991
- 1992
- 1993
- 1994
- 1995
- 1996
- 1997
- 1998
- 1999
- 2000
- 2001
- 2002

Speed (MHz):
- Processor Clock
- Memory Bus Clock
This Talk

- Examine software support to improve locality
  - Compiler analysis
  - Run-time transformations

- Focus on scientific computations
  - Performance matters
  - Easiest to analyze / improve
  - Two types of access patterns
    - Regular
    - Irregular
Outline

- Regular scientific applications
  - Padding transformations
  - Tiling transformations

- Irregular scientific applications
  - Data / computation transformations
  - Adaptive irregular computations
  - Parallel irregular reductions
Regular Scientific Computations

- **Characteristics**
  - Multidimensional arrays
  - Multiple loop nests
  - Also image processing, database scans

- **Regular access patterns**
  - Unit-stride access → exploit spatial locality

**Regular codes**

```
do i = 1, N
  do j = 1, N
    ... = node[j, i]
```
Cache Misses

- **Capacity miss**
  - Data flushed due to lack of overall space

- **Conflict miss**
  - Data flushed due to limited set associativity
  - Can be 50% of cache misses [MT:ASPLOS’96]
  - Severe (recurring) conflict misses between references
  - Solution – data transformations
Data Layout Transformations

- **Padding**
  - Inter-variable - change of base address
    \[
    \text{real A[N], B[N] } \rightarrow \text{ real A[N], DUM[PAD], B[N]}
    \]
  - Intra-variable - change of array dimension
    \[
    \text{real A[N,N] } \rightarrow \text{ real A[N+PAD, N]}
    \]

- **PAD algorithm**
  - Analyze program, predict severe misses
  - Find references with consistent access patterns
    \[
    \text{distance (A[i,j-1], B[i+1,j+1]) = |BaseA - BaseB + } \delta \text{ |}
    \]
  - Pad variables if distance in cache < 1 cache line
Intra-variable Padding

real $A[N,N]$  $\rightarrow$  real $A[N+\text{PAD}, N]$
Inter-variable Padding

real B[N], C[N] → real B[N], DUM[PAD], C[N]

Cache

Cache
Another View of Padding

- Uniformly generated references
- Vertical lines
  - $V[i,j]$
  - base addresses

\[
\begin{align*}
\text{do } j &= 2, N-1 \\
\text{do } i &= 2, N-1 \\
=A[i,j] &
\end{align*}
\]
\[
\begin{align*}
&= B[i,j] + B[i,j+1] \\
&= C[i,j] + C[i,j+1] \\
&= D[i,j] + D[i,j+1]
\end{align*}
\]
PAD Inter-variable Padding

- Pad variables until conflicts eliminated in all loops

- Original layout:
  - Severe conflicts

- PAD layout:
  - Small pads avoid severe conflicts
Group-Temporal Reuse

- Reuse of one reference by another

\[
\begin{align*}
\text{do } & j = 2, N-1 \\
\text{do } & i = 2, N-1 \\
& = A[i, j] \\
& = B[i, j] + B[i, j+1] \\
& = C[i, j] + C[i, j+1] \\
& = D[i, j] + D[i, j+1]
\end{align*}
\]

- To preserve reuse
  - no references under arcs
GROUPPAD Algorithm

- Analyzes group-temporal reuse
- Considers different reference orderings
Impact of PAD on Cache Miss Rates

- Implemented in SUIF compiler
- Reduced misses for some programs

![Cache Miss Rate Chart]

- Original
- Pad
Impact of PAD on Execution Time

- Fewer misses $\rightarrow$ execution time improvements

![Graph showing speedup for various programs on different processors]
Impact of Padding on Cache Misses

- Pad avoids severe conflicts
- GroupPad exploits all reuse possible

![Graph showing impact of padding on cache misses]

CTHPC 2003
Outline

- Regular scientific applications
  - Padding transformations
  - Tiling transformations

- Irregular scientific applications
  - Data / computation transformations
  - Adaptive irregular computations
  - Parallel irregular reductions
Tiling Optimizations

- Move reuses closer in time
- Example: matrix multiplication

```
do J=1,N
  do K=1,N
    do I=1,N
  end do
end do
```

```
do KK=1,N,TK
  do II=1,N,TI
    do J=1,N
      do K=KK,min(KK+TK-1,N)
        do I=II,min(II+TI-1,N)
        end do
      end do
    end do
  end do
end do
```

- Tile data should fit in cache
Avoiding Conflict Misses

- **Need to spread out tile columns on cache**

  ![Diagram of cache conflicts and no conflicts]

- **Targets**
  - 2D linear algebra
  - 3D stencil codes

  ```c
  do k=kk,kk+T
      do j=jj,jj+T
          do i=ii,ii+T
              A[i,j,k] = B[i±1,j±1,k±1]
  ```

CTHPC 2003
Selecting Non-Conflicting Tiles

- **EUC** (extends Euclidean GCD alg)
- Maximal tile sizes
- Recurrences for height and width
  - \( H_i = H_{i-2} \mod H_{i-1} \) (\( H_0 = \text{CacheSize} \) and \( H_1 = \text{ColSize} \))
  - \( W_i = \left\lfloor \frac{H_{i-1}}{H_i} \right\rfloor W_{i-1} + W_{i-2} \) (\( W_{-1} = 0 \) and \( W_0 = 1 \))

- Sample tile dimensions:

- Cache cost model used to choose tile
  - \( \frac{1}{W} + \frac{1}{H} \) (squarish tiles preferred)
Combining Padding with Tiling

- **Padding needed**
  - Leading array dimensions \(\simeq\) cache size
  - Conflicts between nearby array columns

- **Algorithm**
  - Calculate cost of ideal tile (fully associative cache)
  - Iteratively pad array (0,1,2,...)
  - Apply tile size selection (EUC2D, EUC3D)
  - Continue until \(|\text{candidate} - \text{ideal}| < \delta\)

- **Reasoning**
  - Pad size bounded by GCD\(\text{Pad}\)
  - No conflict if GCD(cacheSize, ArrayDim) = TileDim
  - Works well in practice
Tile Cache Utilization

![Graph showing tile cache utilization over time with different cache types and their utilization levels.]

- **euc**
- **lrw**
- **eucPad**
- **lrwPad**

CTHPC 2003
Impact of Tiling on Linear Algebra Codes

- Tile size selection improves performance
- Padding avoids pathological problem sizes

![Graph showing the impact of tiling on linear algebra codes](image_url)
Impact of Tiling 3D Stencils

- Tile size selection improves performance
- Padding improves stability

![Graph showing impact of tiling on Ultrasparc2 performance. The x-axis represents MGRID Resid Problem Size, and the y-axis represents MFlops. The graph compares three cases: Orig, Tile, and Pad. The Tile case shows the most consistent performance across different problem sizes.]
Outline

- Regular scientific applications
  - Padding transformations
  - Tiling transformations

- Irregular scientific applications
  - Data / computation transformations
  - Adaptive irregular computations
  - Parallel irregular reductions
Irregular Scientific Applications

- Scientists use complex models
  - Requires irregular meshes, n-body solvers
Irregular Scientific Computations

- **Application areas**
  - Irregular meshes (computation fluid dynamics)
  - Molecular dynamics (molecular biology)

- **Characteristics**
  - Memory accesses via index array
  - Irregular memory accesses ⇒ poor locality
  - Requires run-time transformations

```
Irregular codes

do i = 1, M
  ... = node[ edge1[i] ]
  ... = node[ edge2[i] ]
```
Observations on Run-time Transformations

◆ Useful for both sequential & parallel codes
  - Improvement = (benefit – run-time overhead)

◆ Optimizations for locality
  - Benefit ⇒ reducing memory accesses
  - Memory access costs moderate (though increasing)
  - Overhead must be kept low

◆ Optimizations for parallelism
  - Benefit ⇒ improving parallelism & communication
  - Parallelism, communication costs high
  - Can afford high overhead transformations
Irregular Scientific Computations

- Irregular memory accesses
  - Index arrays, pointers
  - Poor cache utilization

- Reductions
  - Associative & commutative operations
  - Can be executed in any order
  - Common idiom in irregular computation

```c
do time_step
  do i = 1, n  /* reduction */
    s = ...
    y[ idx1[i] ] += s
    y[ idx2[i] ] -= s
```

Data y: 3 1 5 2 4

Computation i: 3 1 4 2
Graphical Representation of Irregular Codes

real  x[N], y[N]

do time_step = 1, nstep
  do i = 1, E
    p = idx1[i]
    q = idx2[i]
    f = force[x[p], x[q]]
    y[p] = y[p] + f
    y[q] = y[q] - f
  enddo
enddo
Locality Transformations

- Reorder data & computation for cache

- Distribute data & computation to processors
Framework for Locality Reordering

**Framework**

1. Classify computation by # irregular accesses / iteration
2. If single access, reorder computation only
3. Else reorder data, then reorder computation

![Diagram showing computation and data reordering process](image)
Computation Reordering

- Lexicographical sort
  - Sorting computation according to data location

\[ \{ (b,c) \ (d,e) \ (a,b) \ (c,d) \} \rightarrow \{ (a,b) \ (b,c) \ (c,d) \ (d,e) \} \]

location: a b c d e
Data Reordering – Traversal Algorithms

- **Reverse Cuthill-McKee (RCM)**: reverse BFS order
- **Consecutive packing (CPACK)**: first touch order
Data Reordering – Partitioning Algorithms

- Space filling curves (MORTON)
- Recursive coordinate bisection (RCB)
- Multi-level graph partitioning library (METIS) - graph

MORTON
RCB

METIS

CTHPC 2003
Examples of Partitioning Algorithms

RCB

MORTON

Coarsening  Partitioning  Projection

METIS

CTHPC 2003
# Data Reordering Algorithms

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Overhead</th>
<th>Quality</th>
<th>Automation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Traversal</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RCM</td>
<td>Low</td>
<td>Low</td>
<td>OK</td>
</tr>
<tr>
<td>CPACK</td>
<td>Low</td>
<td>Low</td>
<td>OK</td>
</tr>
<tr>
<td><strong>Partitioning</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MORTON</td>
<td>Low</td>
<td>High</td>
<td>coordinate info</td>
</tr>
<tr>
<td>RCB</td>
<td>High</td>
<td>High</td>
<td>coordinate info</td>
</tr>
<tr>
<td>METIS</td>
<td>High</td>
<td>High</td>
<td>OK</td>
</tr>
<tr>
<td><em>Ideal</em></td>
<td>Low</td>
<td>High</td>
<td>OK</td>
</tr>
<tr>
<td><strong>Geometric</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPACK</td>
<td>Low</td>
<td>Low</td>
<td>OK</td>
</tr>
<tr>
<td>RCM</td>
<td>Low</td>
<td>Low</td>
<td>OK</td>
</tr>
<tr>
<td><strong>Graph</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MORTON</td>
<td>Low</td>
<td>High</td>
<td>coordinate info</td>
</tr>
<tr>
<td>RCB</td>
<td>High</td>
<td>High</td>
<td>coordinate info</td>
</tr>
<tr>
<td>METIS</td>
<td>High</td>
<td>High</td>
<td>OK</td>
</tr>
<tr>
<td><em>Ideal</em></td>
<td>Low</td>
<td>High</td>
<td>OK</td>
</tr>
</tbody>
</table>
Low-Overhead Graph Partitioning (GPART)

- Hierarchical clustering with randomly chosen neighbors
- Contiguously place nodes in a partition
- Preserve hierarchy in memory layout
GPART Algorithm

1. **SORT**
   - sort nodes by degree

2. **MERGE**
   - merge neighboring partitions (up to size \( pLimit \))

3. **FILTER**
   - eliminate edges within a partition

4. **LOOP**
   - increase \( pLimit \) by \( growFactor \), repeat

5. **LAYOUT**
   - map partitions according to hierarchy
Outline of GPART algorithm

sort nodes by descending degree

\[ pLimit = \text{cacheLineSize}; \quad \text{maxLimit} = \text{L1\_CacheSize} \]

while \( pLimit \leq \text{maxLimit} \) {

  for each node \( N \) \textit{(in sorted order)} {

    \( P = \) partition containing \( N \)
    \[ \text{if} \ (\text{size}(P) < pLimit) \quad \{ \]
    \text{for each neighbor node} \( M \) \text{of} \( N \) \text{in adj. list} {

      \( Q = \) partition containing \( M \)

      \text{if} \ (P == Q) \text{ mark } M \text{ as merged neighbor}

      \text{else if} \ (\text{size}(P) + \text{size}(Q) \leq pLimit) \text{ } \{ 

      \text{merge } Q \text{ into } P

      \text{mark } M \text{ as merged neighbor}

      \text{if} \ (\text{size}(P) = pLimit) \text{ break (exit for loop)}

    \} \}

    \}

  \}

eliminate merged neighbors in adjacency list

\[ pLimit = pLimit \times \text{growFactor} \quad // \text{new partition size} \]

}\}

recursively lay out nodes according to hierarchy
Compiler Support

- Identify irregular reductions
- Locate access pattern changes
- Insert library call - reorder data & computation

**original code**

```plaintext
#pragma coord(x, c)
idx[ ] = ... // init idx[ ]

do t = 1, time
  if (change)
    idx[...] = ...

  do i = 1, M
    ... = x[ idx[i] ]
```

**transformed code**

```plaintext
idx[ ] = ... // init idx[ ]
inspect(x, idx, c)
do t = 1, time
  if (change)
    idx[...] = ...
    inspect(x, idx, c)

  do i = 1, M
    ... = x[ idx[i] ]
```
Current Implementation in SUIF

- **Fortran-like programs**
  - Handle only array data
  - No pointer analysis

- **Automation is limited**
  - Tested with only small kernels
  - Handle typical irregular reduction
  - No interprocedural analysis
  - Need more analysis to guarantee legality
Experimental Evaluation

◆ Applications
  - Irreg : iterative PDE solver in fluid dynamics
  - NBF : GROMOS molecular dynamics
  - Moldyn : CHARMM molecular dynamics

◆ Input meshes (Edge/Node ≈ 8)
  - foil : 3D mesh of a parafoil
  - auto: 3D mesh of GM’s Saturn
  - mol1: molecule interaction in 3D (small)
  - mol2: molecule interaction in 3D (large)

◆ Extended SUIF compiler
◆ HPC10000 : UltraSparc II (400 MHz, 16KB L1, 4MB L2)
Overhead

<table>
<thead>
<tr>
<th></th>
<th>MORTON</th>
<th>RCB</th>
<th>METIS</th>
<th>GPART</th>
<th>RCM</th>
<th>CPACK</th>
</tr>
</thead>
<tbody>
<tr>
<td>average</td>
<td>1.25</td>
<td>6.44</td>
<td>7.64</td>
<td>2.23</td>
<td>1.10</td>
<td>1.00</td>
</tr>
<tr>
<td>overhead</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Impact on Sequential Performance

- Performance improved by all optimizations
- MORTON best all-around if coordinates are available
- GPART best all-around when coordinates are unknown

![Average Execution Time Graph]

**Average Execution Time**

- MORTON
- RCB
- METIS
- GPART
- RCM
- CPACK

**Number of Iterations**

- 20
- 40
- 80
- 160
- Infinity

**Normalized Exec-Time**

- 0.0
- 0.2
- 0.4
- 0.6
- 0.8
- 1.0
- 1.2
Impact on Cache Performance

**IRREG**

- **L1 foil**
- **L2 foil**
- **L1 auto**
- **L2 auto**

**NBF**

- **L1 mol1**
- **L2 mol1**
- **L1 mol2**
- **L2 mol2**

**MOLDYN**

- **L1 mol1**
- **L2 mol1**
- **L1 mol2**
- **L2 mol2**

**Average**

<table>
<thead>
<tr>
<th></th>
<th>L1</th>
<th>L2</th>
</tr>
</thead>
<tbody>
<tr>
<td>ORIG</td>
<td>20.6%</td>
<td>5.7%</td>
</tr>
<tr>
<td>RCB</td>
<td>6.3%</td>
<td>1.8%</td>
</tr>
<tr>
<td>METIS</td>
<td>6.7%</td>
<td>1.8%</td>
</tr>
<tr>
<td>GPART</td>
<td>7.2%</td>
<td>2.1%</td>
</tr>
<tr>
<td>RCM</td>
<td>9.4%</td>
<td>2.6%</td>
</tr>
<tr>
<td>CPACK</td>
<td>10.4%</td>
<td>3.3%</td>
</tr>
</tbody>
</table>

CTHPC 2003
Overhead of Algorithms

Overhead (MIPS R10000)

Overhead (average)

<table>
<thead>
<tr>
<th>Method</th>
<th>Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>RCB, METIS</td>
<td>21.5</td>
</tr>
<tr>
<td>GPART</td>
<td>5.9</td>
</tr>
<tr>
<td>RCM, CPACK</td>
<td>2.6</td>
</tr>
</tbody>
</table>
Impact on Execution Time

**Avg Improvement at Infinity**

<table>
<thead>
<tr>
<th>Method</th>
<th>(R10K)</th>
<th>(Alpha)</th>
</tr>
</thead>
<tbody>
<tr>
<td>RCB</td>
<td>49.0%</td>
<td>39.3%</td>
</tr>
<tr>
<td>METIS</td>
<td>48.5%</td>
<td>38.6%</td>
</tr>
<tr>
<td>GPART</td>
<td>48.0%</td>
<td>37.6%</td>
</tr>
<tr>
<td>RCM</td>
<td>38.9%</td>
<td>27.6%</td>
</tr>
<tr>
<td>CPACK</td>
<td>29.8%</td>
<td>24.9%</td>
</tr>
</tbody>
</table>
Outline

- Regular scientific applications
  - Padding transformations
  - Tiling transformations

- Irregular scientific applications
  - Data / computation transformations
  - Adaptive irregular computations
  - Parallel irregular reductions
Adaptive Irregular Computations

- Access pattern may change dynamically
  - Reflect changes in underlying computation structure
    - Adaptive mesh refinement
    - Particle movement
  - Implemented as change to index array

<table>
<thead>
<tr>
<th>Irregular codes</th>
<th>Adaptive irregular codes</th>
</tr>
</thead>
<tbody>
<tr>
<td>do i = 1, M</td>
<td>do t = 1, time</td>
</tr>
<tr>
<td></td>
<td>if (change) idx[...] = ...</td>
</tr>
<tr>
<td></td>
<td>do i = 1, M</td>
</tr>
<tr>
<td></td>
<td>... = x[idx[i]]</td>
</tr>
<tr>
<td></td>
<td>... = x[idx[i]]</td>
</tr>
</tbody>
</table>
Adaptive Computation – Characteristics

- **Synthetic adaptive computation**
  - Swap 10% nodes every 10 iterations
  - Performance degrades after access pattern changes
  - Reapplying optimizations maintains performance
Adaptive Codes – Optimization Strategy

- For best overall performance (including overhead)
  - Periodically reapply locality optimization
  - Frequency depends on overhead
  - Legal ⇒ only performance affected
Optimizing Adaptive Computations

- Calculating locality optimization frequency

- Sampling Iterations
  - Measured Information
    - overhead
    - benefit

- Input Program Characteristics
  - number of iterations
  - adaptation frequency
  - % of data changed

- Cost Model
  - which transformation?
  - how often to transform?

- Running Parameters
Adaptive Computation - Cost Model

- Periodically reapply optimization
- Net benefit: \( G(n) = A - n \times \text{Overhead} \)
- Find maximal point of \( G(n) \) (\( n > 0 \)), \( G'(n_0) = 0 \)

![Diagram showing the comparison between original and optimized code](image)
Experimental Validation of Cost Model

- Vertical bars: predicted by cost model
  - maximum gain for each algorithm
On-the-fly Application of Cost Model

- **Cost model parameters**
  - Overhead of each optimization
  - Benefit of each optimization
  - Slowdown rate

- **On-the-fly algorithm**
  - Choose low-overhead optimization (GPART)
  - Measure execution time every iteration
  - Determine performance slowdown rate
  - Continuously calculate optimization frequency
Precision of On-the-fly Application

- On-the-fly cost model matches with static cost model

![Cost Model Comparison Diagram]

- Performance Gain (%) for different applications:
  - Foil: 3% and 4%
  - Auto: 4% and 4%
  - Mol1: 5% and 4%
  - Mol2: 6% and 6%

Number of transformations applied:
- IRREG: 4
- NBF: 5
- MOLDYN: 6
Outline

- **Regular scientific applications**
  - Padding transformations
  - Tiling transformations

- **Irregular scientific applications**
  - Data / computation transformations
  - Adaptive irregular computations
  - Parallel irregular reductions
Parallel Irregular Reduction

- Irregular reduction
  - Many parallelization strategies
    - Array privatization
    - Selective privatization
    - Synchronized write
    - Local write (accounting for locality)

Irregular reduction

```c
for (i = 0; i < M; i++) {
    f = fn ( x[idx[i]], x[idx2[i]] )
    y[idx[i]] += f
    y[idx2[i]] += -f
}
```

Data access

Computation (=iteration)

Data (x[ ], y[ ])

CTHPC 2003
Array Privatization (ReplicateBufs)

- Replicate entire array as local buffers
- Combine buffer results even for unused elements
- Synchronization overhead

![Diagram showing array privatization process]

X : unused element
Selective Privatization  (SelectPrivate)

- Avoid replication unused elements
- Inspector
  - Select local and boundary iterations
  - Construct private copies

![Diagram showing the process of privatization with nodes and arrows indicating access via hashing and collection of private copies.]

1. P1
   - 3
   - 1
   - 2

2. P2
   - 1
   - 4
   - 2
   - 3
   - 4

3. P3
   - 2
   - 4
   - 5

: private copy
: access via hashing
: collect private copies
Synchronized Write (SynchWrite)

- Group iterations by processor distance
- Low utilization of processors (less than half)
- Multiple synchronized phases (2*p-2 phases)
Locality-Conscious Distribution (LocalWrite)

- Compute and update only locally-owned data
  - No synchronization needed
  - Compiler inserts inspector
    - Apply locality transformations
    - Select local and boundary iterations for LocalWrite
  - Boundary iteration – replicate computation

: replicated iteration
Implementation of Local Write

- **DWA-LIP [Euro-Par’99]**
  - Original index array
  - Prefetch array `next`
  - Extra array access at each iteration

- **LocalWrite [PACT’98]**
  - Modify index array `idx2`
  - Reorders iterations
  - Direct access

```c
Inspector(next)
ii = next[0]
for (i = 0; i < count, i++) {
    y[idx[ii]] += ...
    ii = next[i];
}

Inspector(idx2)
for (i = 0; i < count, i++) {
    y[idx2[i]] += ...
}
```
# Comparison of Parallel Reduction Scheme

<table>
<thead>
<tr>
<th></th>
<th>Replicate Data</th>
<th>Synch</th>
<th>Replicate Comp</th>
<th>Inspector</th>
<th>Scalable</th>
</tr>
</thead>
<tbody>
<tr>
<td>ReplicateBufs</td>
<td>All</td>
<td>✔</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SelectPrivate</td>
<td>Some</td>
<td>✔</td>
<td></td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>SynchWrite</td>
<td></td>
<td>✔</td>
<td></td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>LocalWrite</td>
<td></td>
<td>✔</td>
<td></td>
<td>✔</td>
<td>✔</td>
</tr>
</tbody>
</table>
Experiment

- **SUN HPC 10000**
  - Use 4, 8, 16 processors out of 64 processors
  - 400 Mhz UltraSparc II

- **Extended SUIF compiler**
  - Automatic generation for LocalWrite

- **Application kernels**
  - Irreg: computational fluid dynamics
  - NBF, Moldyn: molecular dynamics

- **Inputs (70K nodes from Moldyn)**
  - Connectivity : Edges/Nodes (E/N) = 6 — 124
  - Locality : %local iterations = 0% — 90%
  - Adaptivity : adaptation interval = 10 — infinity
Comparison with Commercial Compilers

- Insufficient support for irregular reductions

![Graph comparing SUNW's pro and ReplicateBuf](#)
Comparison of LocalWrite Implementation

- LocalWrite improves 56% over DWA on average

![DWA vs LocalWrite speedups (16 procs)]

![DWA vs LocalWrite overhead (seconds)]
Base Speedup

- **Base Inputs**
  - E/N = 124, %local iteration ≈ 30%, non-adaptive

Average Speedup (Base)
Impact of Connectivity

- %local iteration 30%, non-adaptive

Average Speedup vs Connectivity

- ReplicateBuf
- SelectPrivate
- SynchWrite
- LocalWrite
Impact of Locality

- E/N = 9 or 124, non-adaptive
Impact of Adaptivity

- $E/N = 124$, %local iterations 30%

**Average Speedup vs Adaptivity**

- ReplicateBufs
- SelectPrivate
- SynchWrite
- LocalWrite

**Inspector overhead**

<table>
<thead>
<tr>
<th>Type</th>
<th>Overhead (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ReplicateBufs</td>
<td>0.0</td>
</tr>
<tr>
<td>Select-Private</td>
<td>3.0</td>
</tr>
<tr>
<td>Synch-Write</td>
<td>3.4</td>
</tr>
<tr>
<td>Local-Write</td>
<td>1.4</td>
</tr>
</tbody>
</table>
Impact On Parallel Performance

- **Two methods to parallelize irregular reductions**
  - Locality optimizations improves parallel codes
  - LocalWrite benefits more than ReplicateBuffer

![Graph showing speedups for different methods](image)

**ReplicateBuffer**

**LocalWrite**
Related Work – Regular Computations

- Tiling [Wolf & Lam, PLDI’91]
- Tile self-interference [Lam+, ASPLOS’91]
- Euclidean algorithm [Coleman & McKinley, PLDI’95]
- 3D stencil codes [Weiss+, SC’99]
- Padding single loop nests [Bacon+, CASCON’94]
- Limited global transformations [Manjikian+ TPDS’97]
- Array fusion [Ding & Kennedy, LCPC’99]
- Analytical models [Ghosh+, ASPLOS’98]
- Page mapping [Bugnion+, ASPLOS’96]
Related Work – Irregular Computations

- Lexicographical sort, RCM reordering [Das+ ASME’92]
- METIS, BFS [Al-Furaih+ IPPS’98, Karypis+ SC’95]
- CPACK [Ding & Kennedy, PLDI’99]
- Space filling curve [Mellor-Crummey+, ICS’99]
- RCB reordering [Berger & Bokhari, TOC’87]
- Bucket Sort [Mitchell+, PACT’99]
- Array privatization [ SUIF, Polaris ]
- Selective privatization [ Yu & Rauchwerger, ICS’00]
- Synchronized write [ Gutierrez+, ICS’00]
- DWA-ILP [ Gutierrez+, Euro-Par’99]
- Local write [ Han & Tseng, PACT’98]
Summary

◆ Data & computation transformations
  - Can improve locality
  - Applicable for both regular & irregular codes
    ● Irregular applications requires run-time support
  - Impact can be substantial

◆ Future directions
  - Apply to integer / pointer codes
  - Much more difficult
    ● Less amenable to compiler analysis
    ● Bigger, more complex programs