Optimising Locality of Reference

Cosmin E. Oancea cosmin.oancea@diku.dk

Department of Computer Science (DIKU)
University of Copenhagen

December 2019 PFP Lecture Slides

Motivation

What techniques aimed at optimising locality have we studied?

Motivation

What techniques aimed at optimising locality have we studied?

We covered the case of one perfect-loop nest with affine accesses:

- fusion/fission
- loop interchange,
- tiling (loop stripmining + interchange)

Example: Loop Interchange Enhances Locality of Reference

```
// Bad locality both GPU & CPU
DOALL j = 1, N-1 // grid
  DOALL i = 0, N-1 // block
    A[i,j] = sqrt(A[i,j] + B[i,j]);
  ENDDO
ENDDO
```

```
// Good locality both GPU & CPU
DOALL i = 0, N-1
  DOALL j = 1, N-1
    A[i,j] = sqrt(A[i,j] + B[i,j]);
  ENDDO
ENDDO
```

But a program is a composition of loop nests &
accesses are not always affine &
how about communication in distributed programs (?!)

1. Tiling Affine Loop Nests: Optimising Communication & Load Balancing [Reddy and Bondhugula'14]

2. Iteration and Data Reordering for Loop Nests with Irregular Accesses [Ding and Kenedy'99], [Strout et. al. 2003,2004]
   - Data Reordering (Packing)
   - Iteration Reordering (Locality Grouping)
   - Generalization: Temporal & Spatial Locality HyperGraphs

3. Other Locality Optimizations: Parallel Tracing

A Simple Composition of Two Loop Nests

Effective Automatic Data Allocation for Parallelization of Affine Loop Nests, Chandan Reddy and Uday Bondhugula, ICS 2014.

Running Example: ADI benchmark

```
//forward x sweep
for (i=0; i<N; i++)  //parallel
    for (j=1; j<N; j++) // sequential
S1      X[i,j] -= X[i,j-1]*..;

//upward y sweep
for (j=0; j<N; j++) // parallel
    for (i=1; i<N; i++) // sequential
S2      X[i,j] -= X[i-1,j]*..;
```

- Each loop (nest) can be efficiently parallelized individually,
- in a distributed setting with NO intra-loop-nest communication.
- How about the whole program?

Mapping Available Parallelism to A Set of Nodes

Commonly used patterns for distributing iterations across processors:

- Block Distribution: loop iterations are divided into number-of-processor, nearly-equal contiguous chunks.
- Cyclic Distribution: one iteration to each processor in a round robin fashion. Better load balancing when iterations have non-uniform cost.
- Block-Cyclic Distribution: like cyclic but contiguous chunks of iterations are distributed in round-robin fashion.
- Sudoku Distribution: assigns for example 2-dim tiles to processors such that ALL tiles inside a row or column are mapped to distinct processors.

OpenMP:

```
#pragma omp parallel for schedule(kind [,chunk size])
```

- Block: `schedule(static)`
- Cyclic: `schedule(dynamic)`
- Block-cyclic: `schedule(dynamic, block_size)`

Mapping Available Parallelism to A Set of Nodes

We call a computation mapping (iterations to nodes) optimal if it leads to the lowest communication and perfect load balance.

ADI program: Tiled Version (Tile Size: 128)

```
// forward x sweep
for (jj=0; jj<N; jj+=128) //serial loop
 for (ii=0; ii<N; ii+=128) //parallel loop
  for (i=max(1,ii); i<min(ii+127,N); i++)
   for (j=max(1,jj); j<min(jj+127,N); j++)
S1   X[i][j] -= X[i][j-1]*..;

// upward y sweep
for (ii=0; ii<N; ii+=128) //serial loop
 for (jj=0; jj<N; jj+=128) //parallel loop
  for (j=max(1,jj); j<min(jj+127,N); j++)
   for (i=max(1,ii); i<min(ii+127,N); i++)
S2   X[i][j] -= X[i-1][j]*..;
```

- A node consists of a set of processor operating in shared memory (communication is required between nodes).
- optimal mapping for the forward sweep is block distribution along ii,
- optimal mapping for the upward sweep is block distribution along jj,
- these mappings are not optimal for the entire program, since the transpositon of X requires a lot of communication!

Solution: model the optimal-mapping problem as a graph paritionaing problem on the inter-tile communication graph (TCG).

Inter-Tile Communication Graph (TCG)

- Each Vertex in TCG represents a computation tile.
- An edge e is added between two vertices *iff* there is communication between those tiles (assuming they are executed on different nodes).
- The weight of an edge, e_w is equal to the communication volume between two tiles.

Finding the optimal mapping is equivalent to partitioning TCG into number-of-nodes p equal-sized partitions, i.e., optimal load balancing, with the objective to minimize the sum of those weights that straddle partitions.

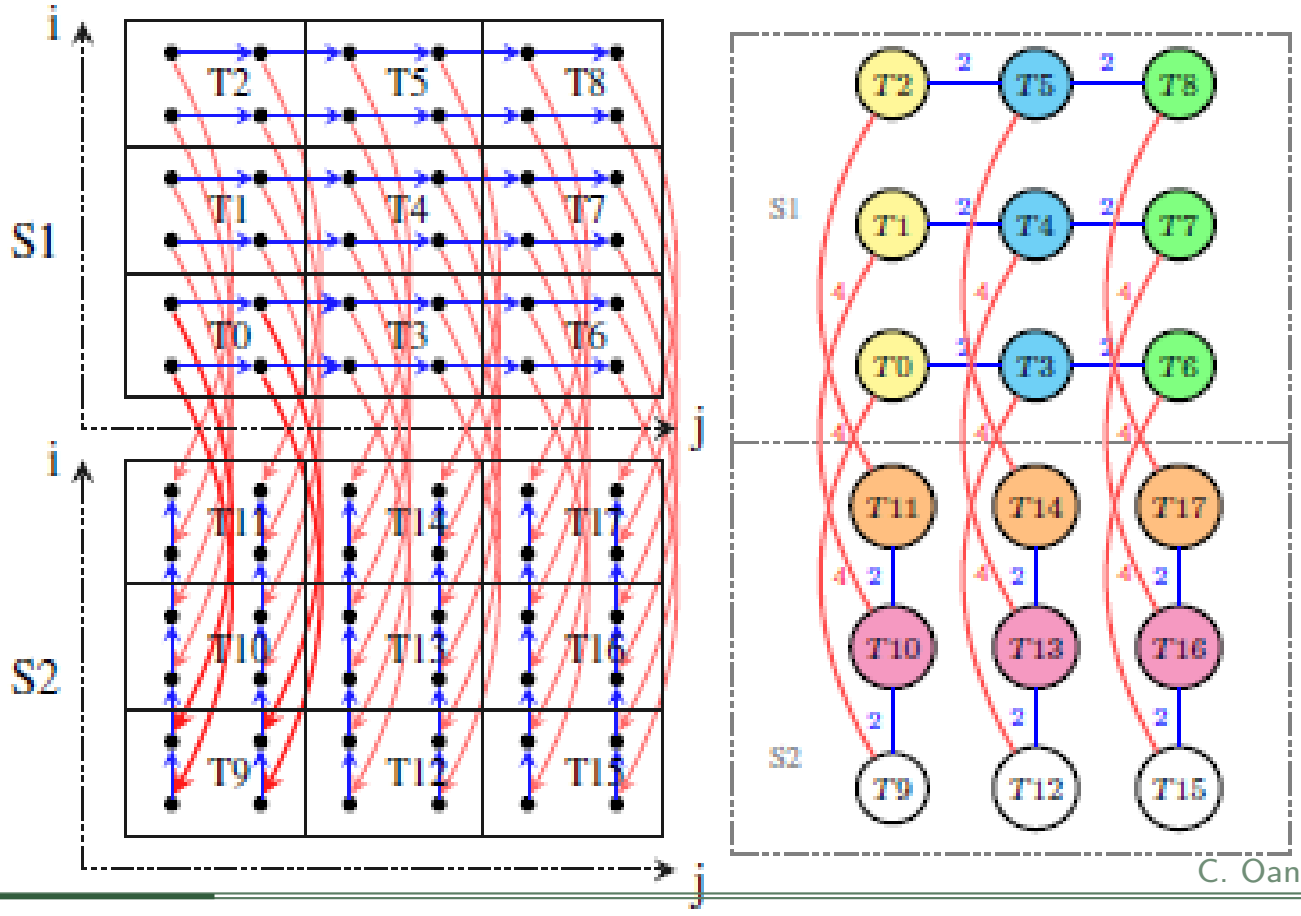
Objective function is the total communication value for entire program execution, under load balancing constraint.

TCG for Running Example

Left: Tiled Iteration Space with Dependencies. Dependence edges that cross tile boundaries used to determined the communication sets.

Right: Corresponding Inter-Tile Communication Graph

- Same color tiles can be executed in parallel!



Load Balancing TCG Constraints

Program consists of multiple parallel phases.

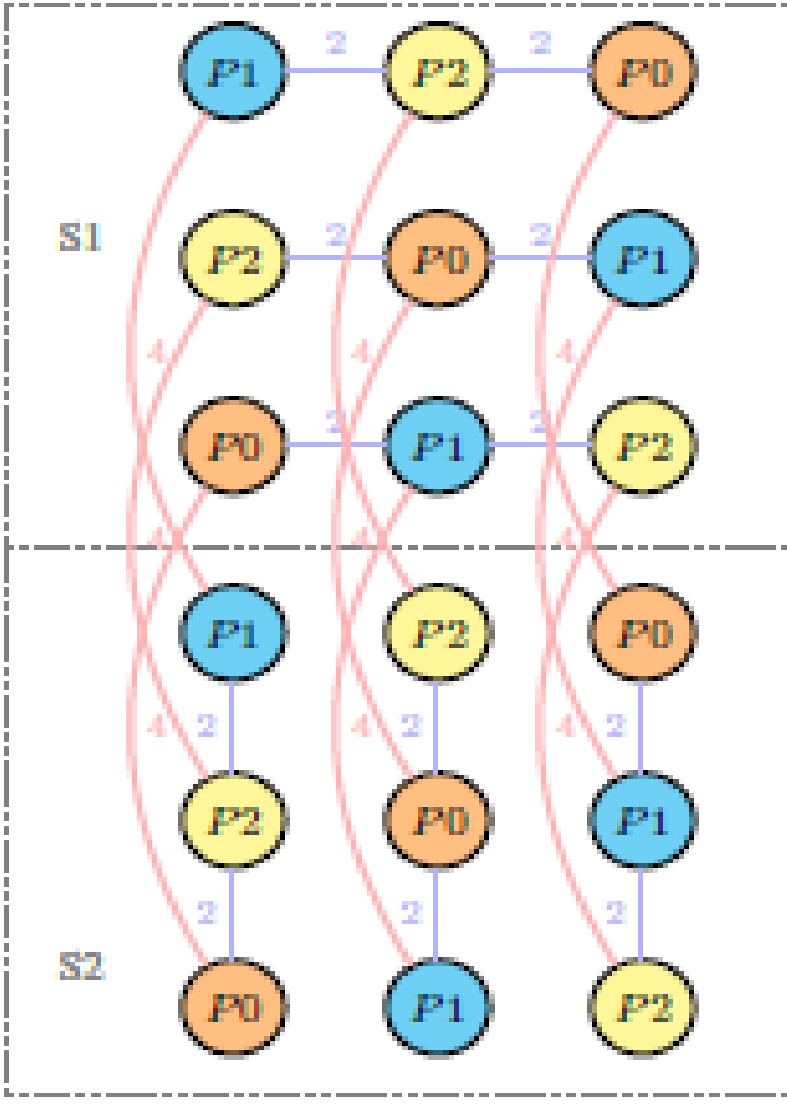
Good load balance $\Rightarrow$ nearly equal number of tiles are allocated to all nodes in each parallel phase.

Add constraints to minimize load imbalance in each parallel phase:

- vertex weights used to distinguish between tiles used in different parallel phases.
- all tiles belonging to parallel phase `i` will have at the $\texttt{i}^{th}$ position the number of iterations in the tile, and the others 0.
- Let S_i^n be the sum of the $\texttt{i}^{th}$ vertex weight component of all vertexes in partition n.
- $\forall i$ vertex weight components, load-balancing constraints are added to minimize the difference between any two partitions n and m, i.e., minimize $\sum_i(\sum_{n \neq m}(S_i^n - S_i^m))$.

TCG Partitioning Result for Running Example

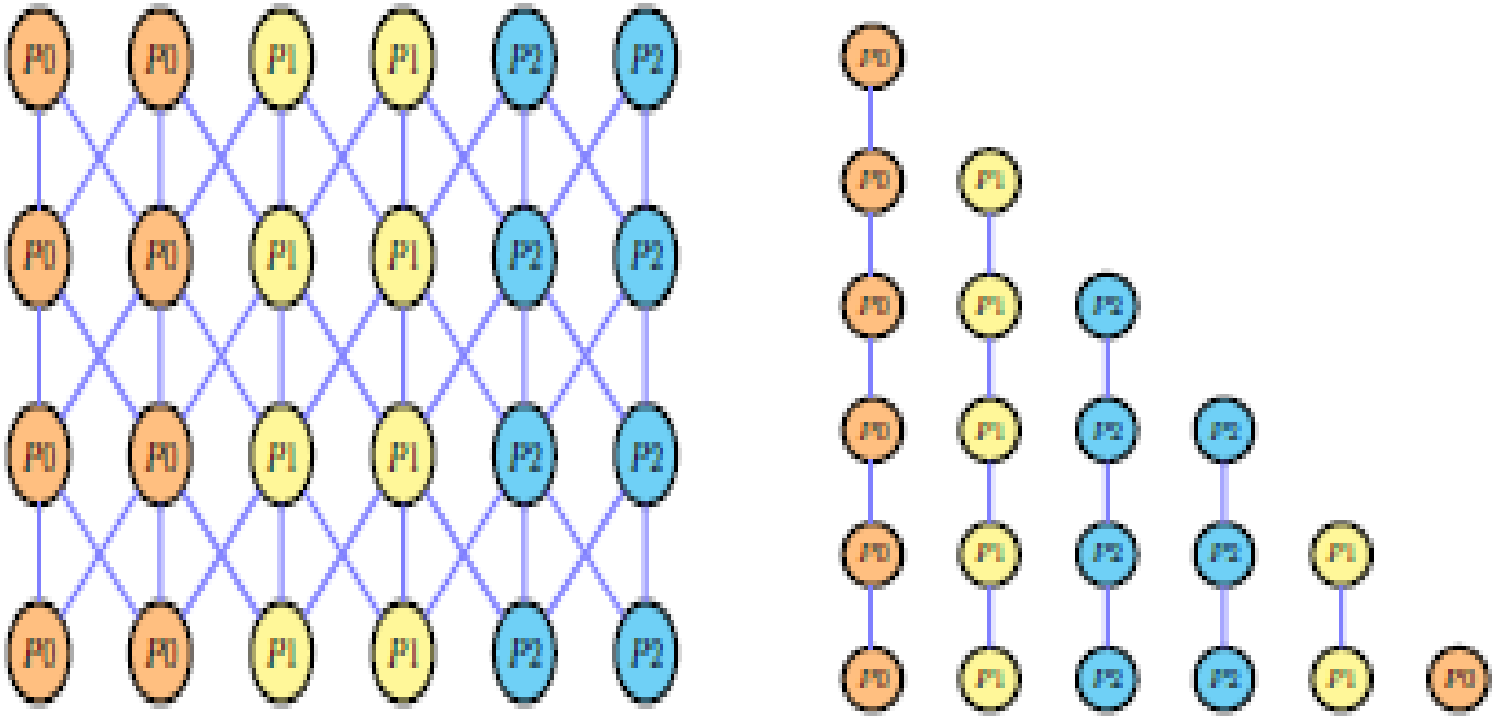
- optimal solution obtained via graph partitioning,
- same colored tile execute on the same processor,
- in each parallel phase, equal number of tiles assigned to each node ⇒ perfect load balance
- computation mapping is identical for both loop nests, i.e., the "expensive" communication (of weight 4) has been eliminated,
- Sudoku distribution is optimal.



TCG Partitioning Result for Other Programs

Left: stencil computation with nearest neighbor communication ⇒ optimal results is block distribution.

Right: unbalanced computation. Result is slightly different from block-cyclic mapping in that first and last column are mapped to P0 instead of first and fourth.



Perfect! Any Difficulties Left?

- As problem size increases, so do the number of vertexes and edges in the graph, and the number of constraints.
- even state-of-the-art graph partitioning software do not scale (heuristics way of solving the NP hard problen),
- for example, METIS takes 240s to partition ADI with 64 vertices into 4 partitions.
- further problem-size increase $\Rightarrow$ drastic decrease in performance, and in the accuracy of the solution, e.g., perfect sudoku mappings were not obtained for more than 32 vertices.

To make the approach scale to larger sizes $\Rightarrow$ an approximation of TCG is computed for a small number of iteration and the result is expanded across the whole iteration space.

Works in practice because accesses are affine, i.e., regular.

Empirical Results: Weak Scaling

Weak Scaling: how the solution time varies with the number of processors for a fixed problem size per processor. Ideally one gets a horizontal line.

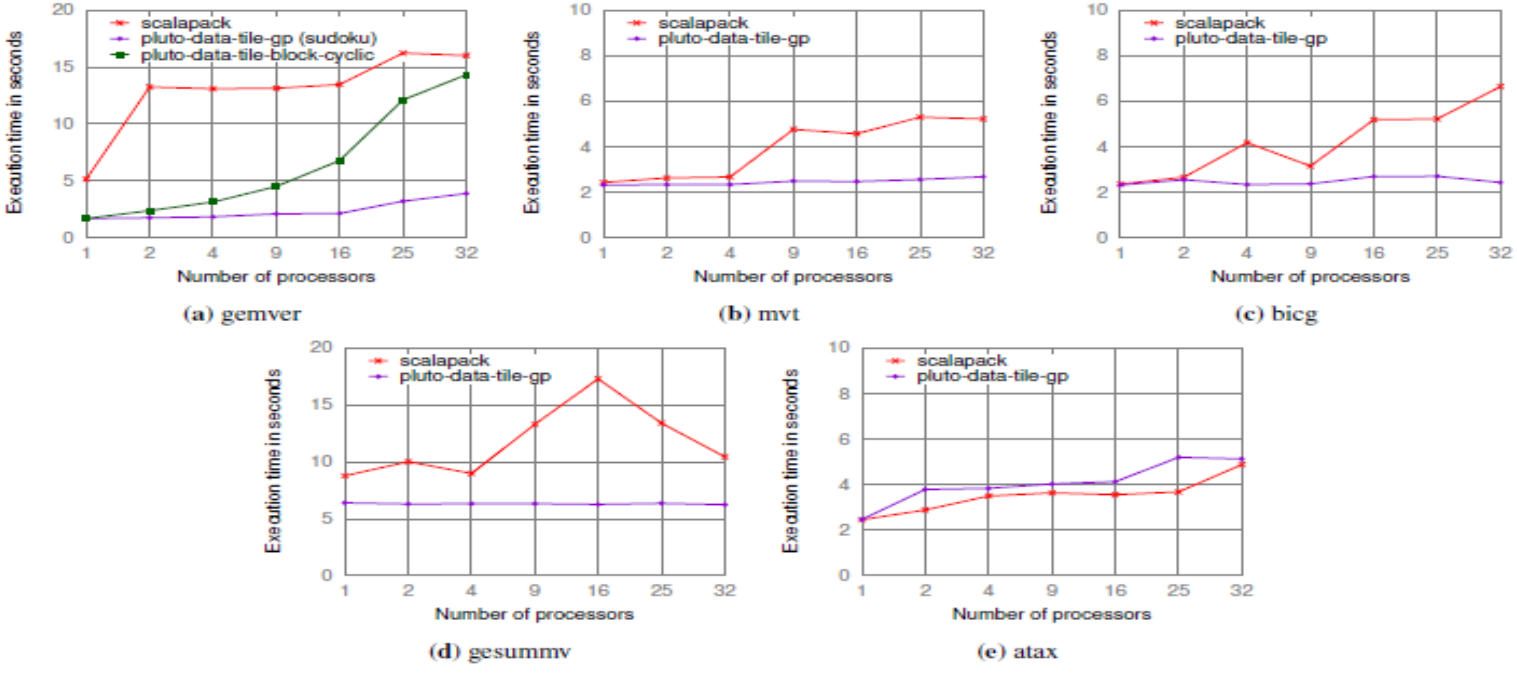
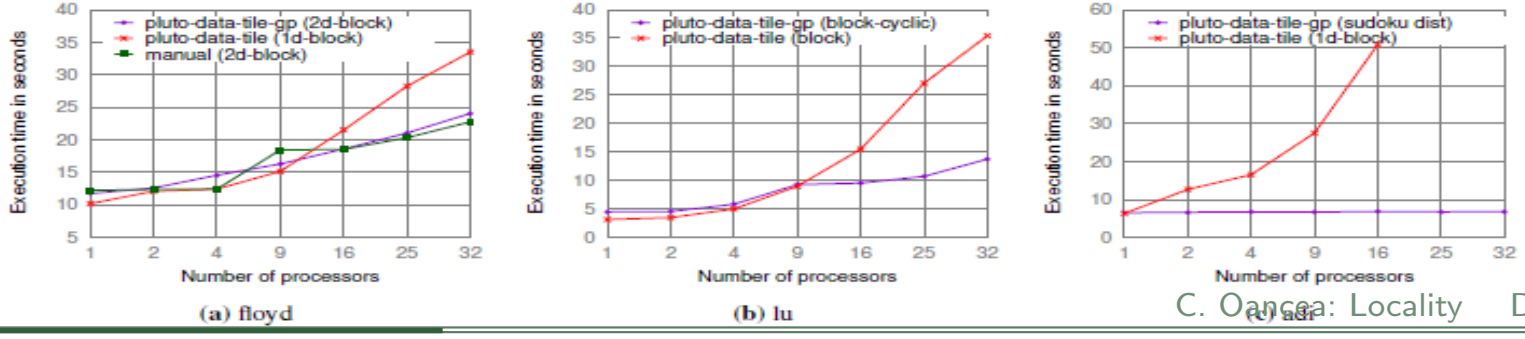


Figure 12: Weak scaling performance of scalapack and pluto-data-tile



1. Tiling Affine Loop Nests: Optimising Communication & Load Balancing [Reddy and Bondhugula'14]

2. Iteration and Data Reordering for Loop Nests with Irregular Accesses [Ding and Kenedy'99], [Strout et. al. 2003,2004]
   - Data Reordering (Packing)
   - Iteration Reordering (Locality Grouping)
   - Generalization: Temporal & Spatial Locality HyperGraphs

3. Other Locality Optimizations: Parallel Tracing

Irregular Computation Based on Indirect Arrays

Improving Cache Performance in Dynamic Applications through Data and Computation Reorganization at Run Time, Chen Ding and Ken Kennedy, PLDI'99

Metrics and Models for Reordering Transformations, Michelle Strout and Paul Hovland, MSP'04.

Irregular applications do not access memory in a strided fashion, e.g.,

- molecular dynamics simulation, which model the movement of particles in some physical domain $\Rightarrow$ the distribution of molecules is unknown until runtime, and even there it changes dynamically.
- sparse linear algebra, e.g., sparse matrix-vector multiplication,
- Impossible to optimise locality of reference statically!
- Use inspector-executor techniques [Saltz et. al] $\Rightarrow$ insert code that reorganize at runtime the order in which iterations are executed or the data layout.

Running Example

- Indirect arrays `left` and `right` are invariant to the outermost loop, hence the runtime iteration/data reordering can be amortized across multiple executions.
- CHARMM, GROMOS, MESH benchmarks.

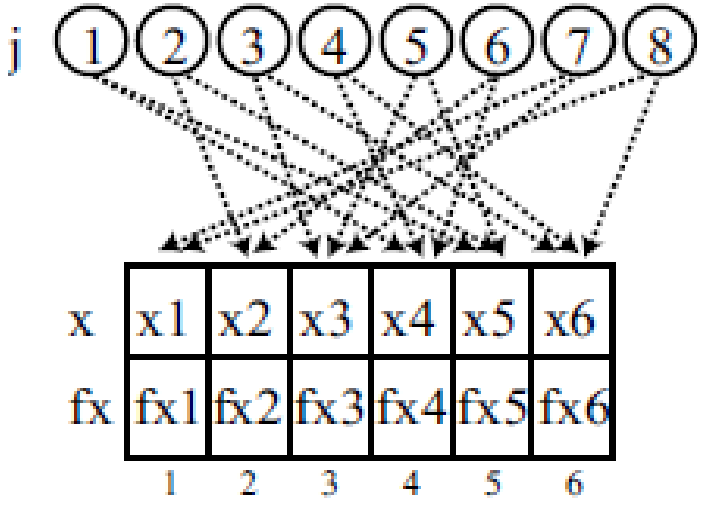
Simplified Moldyn Example: Iteration over the Graph Edges

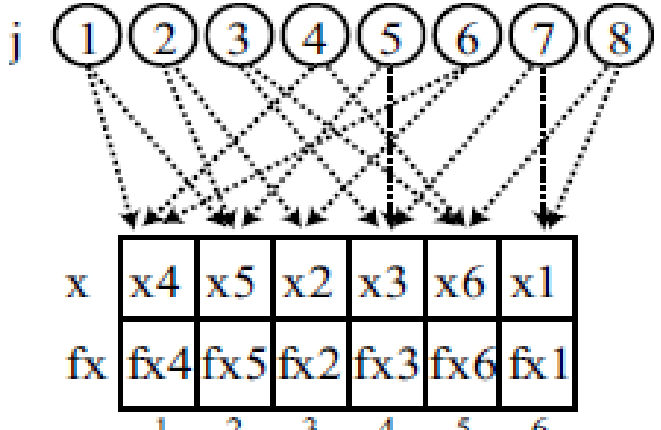
```
  DO s = 1 to num_steps
    // Update location based on old position, velocity and acceleration
    DO i = 1 to num_nodes //parallel
S1    x[i] += vx[i] + fx[i]
    ENDDO
    // Update the forces on the molecule
    DO j = 1 to num_iteractions
S2    fx[ left [j] ] += calcF(x[left[j]], x[right[j]])
S3    fx[ right[j] ] += calcF(x[left[j]], x[right[j]])
    ENDDO
    // Update velocity based on force (acceleration)
    DO k = 1 to num_nodes //parallel
S4    vx[k] += fx[k];
    ENDDO
ENDDO
```

Runtime Data Reordering

- Aims to improve the spatial locality in the loop by reordering the data based on the order in which it is referenced in the loop.

- Iteration `j` accesses `x[left[j]]`, `x[right[j]]`, `fx[left[j]]`, `fx[right[j]]`.
- Top Figure shows the original access patters.
- Bottom Figure shows the access pattern after consecutive-packing (CPACK), i.e., data is repacked to match the order in which it is used in the original loop.
- Notice better spatial locality!





Consecutive Packing (CPACK) Implementation

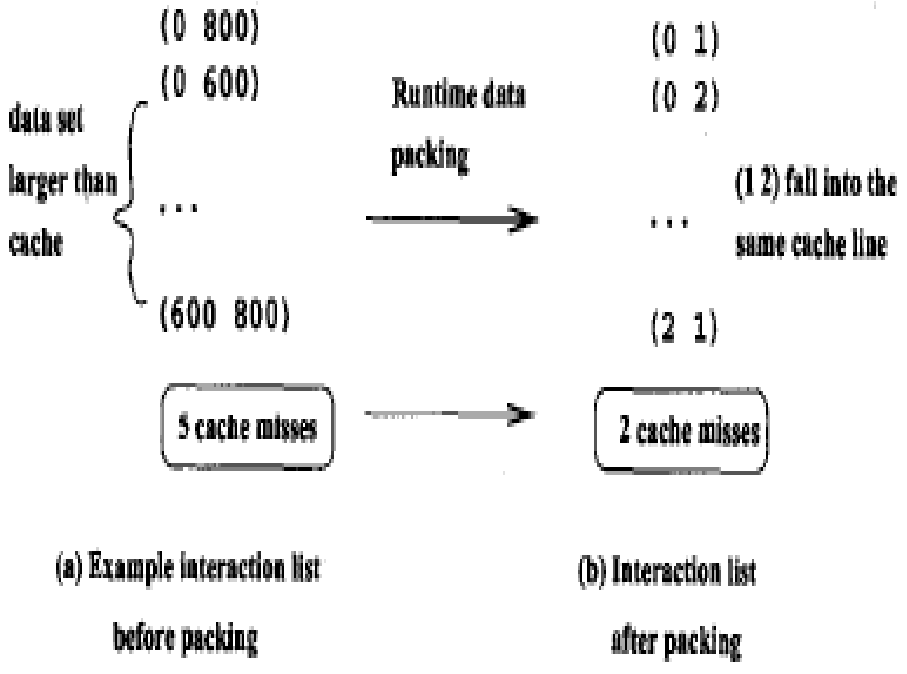
- Aims to improve the spatial locality in the loop by reordering the data based on the order in which it is referenced in the loop.

```
CPACK(left, right)  // Output: σ^-1
// alreadyOrdered bit vector set to 0s.
count = 0
DO j = 1 to num_interactions
  mem_loc1 =  left[j]
  mem_loc2 = right[j]

  IF not alreadyOrdered[mem_loc1]
    σ^-1[count] = mem_loc1
    alreadyOrdered[mem_loc1] = 1
    count = count + 1
  ENDIF
  // DO THE SAME FOR mem_loc2!
ENDDO

DO i = 1 to num_nodes
  IF not alreadyOrdered[i]
    σ^-1[count] = i
    count = count + 1
ENDIF ENDDO
```

Assuming a cache line holds 3 words



Consecutive Packing (CPACK) Implementation

`num_interactions=8, num_nodes = 6`

Orignial indirect indexing for j $= 1 \ldots 8$.

j	1	2	3	4	5	6	7	8
left[j]	4	2	3	4	3	2	1	1
right[j]	5	5	6	6	5	4	3	6

CPACK will result in the following σ^{-1}, where i $= 1 \ldots 6$.

Consecutive Packing (CPACK) Implementation

`num_interactions=8, num_nodes = 6`

Orignial indirect indexing for j $= 1 \ldots 8$.

j	1	2	3	4	5	6	7	8
left[j]	4	2	3	4	3	2	1	1
right[j]	5	5	6	6	5	4	3	6

CPACK will result in the following σ^{-1}, where i $= 1 \ldots 6$.

i	1	2	3	4	5	6
σ^{-1}	4	5	2	3	6	1

σ can be obtained by: `scatter (replicate 6 0)` σ^{-1} `[1..6]`

Consecutive Packing (CPACK) Implementation

`num_interactions=8, num_nodes = 6`

Orignial indirect indexing for j $= 1 \ldots 8$.

j	1	2	3	4	5	6	7	8
left[j]	4	2	3	4	3	2	1	1
right[j]	5	5	6	6	5	4	3	6

CPACK will result in the following σ^{-1}, where i $= 1 \ldots 6$.

i	1	2	3	4	5	6
σ^{-1}	4	5	2	3	6	1

σ can be obtained by: `scatter (replicate 6 0)` σ^{-1} `[1..6]`

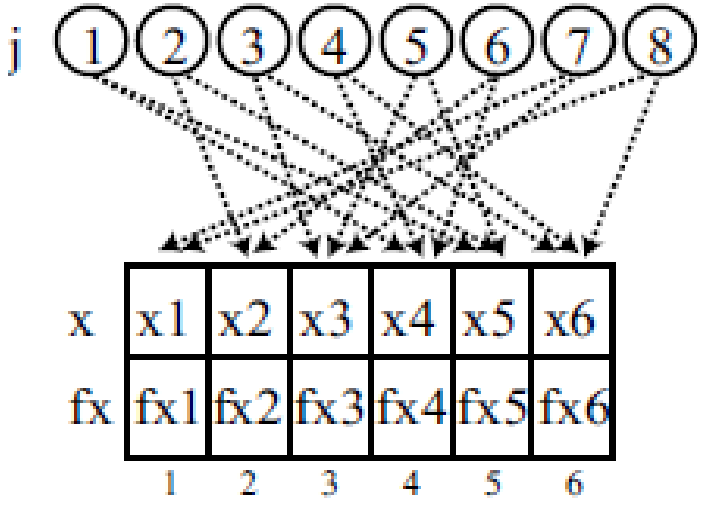
i	1	2	3	4	5	6
σ	6	3	4	1	2	5

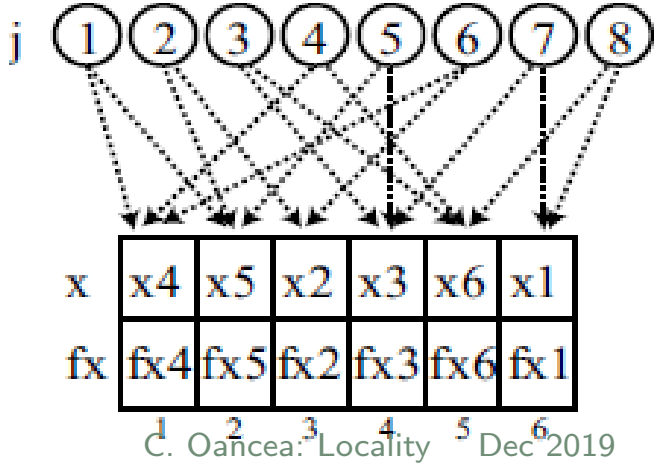
Code After Consecutive Packing

- Aims to improve the spatial locality in the loop by reordering the data based on the order in which it is referenced in the loop.

```
σ⁻¹ = CPACK(left, right)
DO i = 1 to num_nodes
   x'[i] =  x[σ⁻¹[i]]
  fx'[i] = fx[σ⁻¹[i]]
ENDDO
σ = inverse(σ⁻¹) // permute σ⁻¹ [1..n]

DO s = 1 to num_steps
  DO i = 1 to num_nodes  // parallel
    x'[σ[i]] += vx[i] + fx'[σ[i]]
  ENDDO
  DO j = 1 to num_iteractions
    fx'[σ[left [j]]]+=calcF(x'[σ[left[j]]],
                            x'[σ[right[j]]])
    fx'[σ[right[j]]]+=...
  ENDDO
  DO k = 1 to num_nodes // parallel
    vx[k] += fx'[σ[k]]
  ENDDO
ENDDO
```





Optimising Overheads of Data Reordering/Packing

Overhead of (dynamic) data reordering/packing:

- Overhead of data reorganization, e.g., CPACK. This can be amortized over multiple computation iterations.
- Indirection Overhead very expensive @every access:
  - instructional overhead of indirection
  - indirection overhead: one extra load to memory
  - spatial locality might have been compromised in other loops.

```
// Pointer Update
// left' = σ ⊙ left, right' = σ ⊙ right
DO j = 1 to num_iteractions
  fx'[left' [j]]+=calcF(x'[left'[j]],
                        x'[right'[j]])
ENDDO
```

- Pointer Update Optim eliminates the extra load from memory by computing (once) $\sigma \odot$ `left` or `right`

Optimising Overheads of Data Reordering/Packing

```
DO k = 1 to num_nodes // parallel
  vx[k] += fx'[σ[k]]
ENDDO
    ↓ reorganize vx ↓
DO i = 1 to num_nodes // amortized
  vx'[i] = vx[σ^-1[i]]
ENDDO
DO k = 1 to num_nodes // parallel
  vx'[σ[k]] += fx'[σ[k]]
ENDDO
    ↓ parallel loop ⇒ reorder iters ↓
DO k = 1 to num_nodes // parallel
  vx'[k] += fx'[k]
ENDDO
```

- Array Alignment Optim reorganizes vx array in the same way as fx' (and x').
- Legality Requirements:
  1. the range of loop iterations is identical to the range of remapped data.
  2. the loop is parallel (so that its iterations can be reordered).

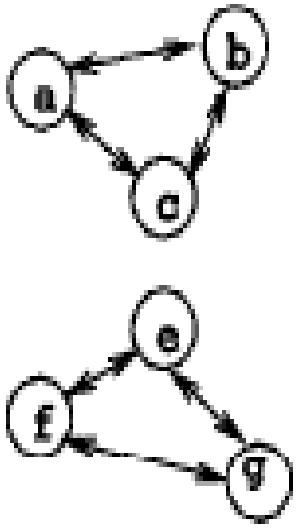
Code After Dynamic Data Reordering/Packing

```
σ^-1 = CPACK(left, right)
σ = inverse(σ^-1)
DO i = 1 to num_nodes // Overhead
   x'[i] =  x[σ^-1[i]]
  fx'[i] = fx[σ^-1[i]]
  vx'[i] = vx[σ^-1[i]]
ENDDO
DO s = 1 to num_steps // convergence loop, allows amortization
  DO i = 1 to num_nodes  // parallel
    x'[i] += vx'[i] + fx'[i]
  ENDDO
  DO j = 1 to num_iteractions
    fx'[left' [j]]+=calcF(x'[left'[j]], x'[right'[j]])
    fx'[right'[j]]+=calcF(x'[left'[j]], x'[right'[j]])
  ENDDO
  DO k = 1 to num_nodes // parallel
    vx'[k] += fx'[k]
  ENDDO
ENDDO
DO i = 1 to num_nodes // Overhead, appears only if x, fx, vx are live
   x[i] =  x'[σ[i]]
  fx[i] = fx'[σ[i]]
  vx[i] = vx'[σ[i]]
ENDDO
```

Iteration Reordering

- Aims to improve the temporal locality across consecutive loop iterations.
- by ordering the iterations that touch the same data item consecutively in the resulted schedule.

Assuming a 3-word cache (cache line = one word)



(a) Example Interactions

(b c)
(e g)
(e f)
(a b)
(f g)
(a c)

(b) Example Sequence
10 misses
(3-element cache, fully associative, LRU replacement)

(a b)
(a c) > Group on (a)

(b c) — Group on (b)

(e g)
(e f) > Group on (e)

(f g) — Group on (f)

(c) Sequence after Locality Grouping
6 misses

Data Packing and Iteration Reordering

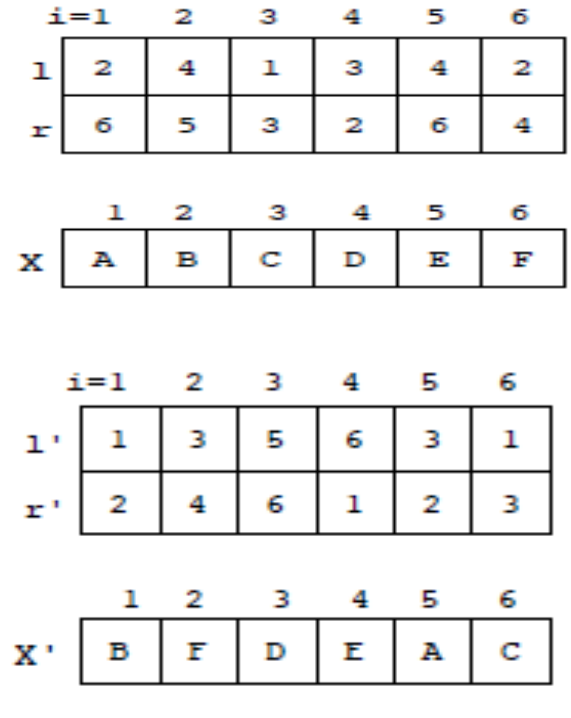
1. Original code below corresp. to the top Figure →

```
DO i = 1 to N
  ... X[l[i]] ...
  ... X[r[i]] ...
ENDDO
```

2. After data reordering/packing, l'=$\sigma\odot$l, r'=$\sigma\odot$r and X', the reorganized X, are shown in middle Figure →

3. Loop iterations are reordered by lexicographically sorting index arrays l' and r' into l'' and r'', as shown in bottom Figure →

```
DO i = 1 to N
  ... X'[l''[i]] ...
  ... X'[r''[i]] ...
ENDDO
```



Empirical Evaluation: Effects of Reordering

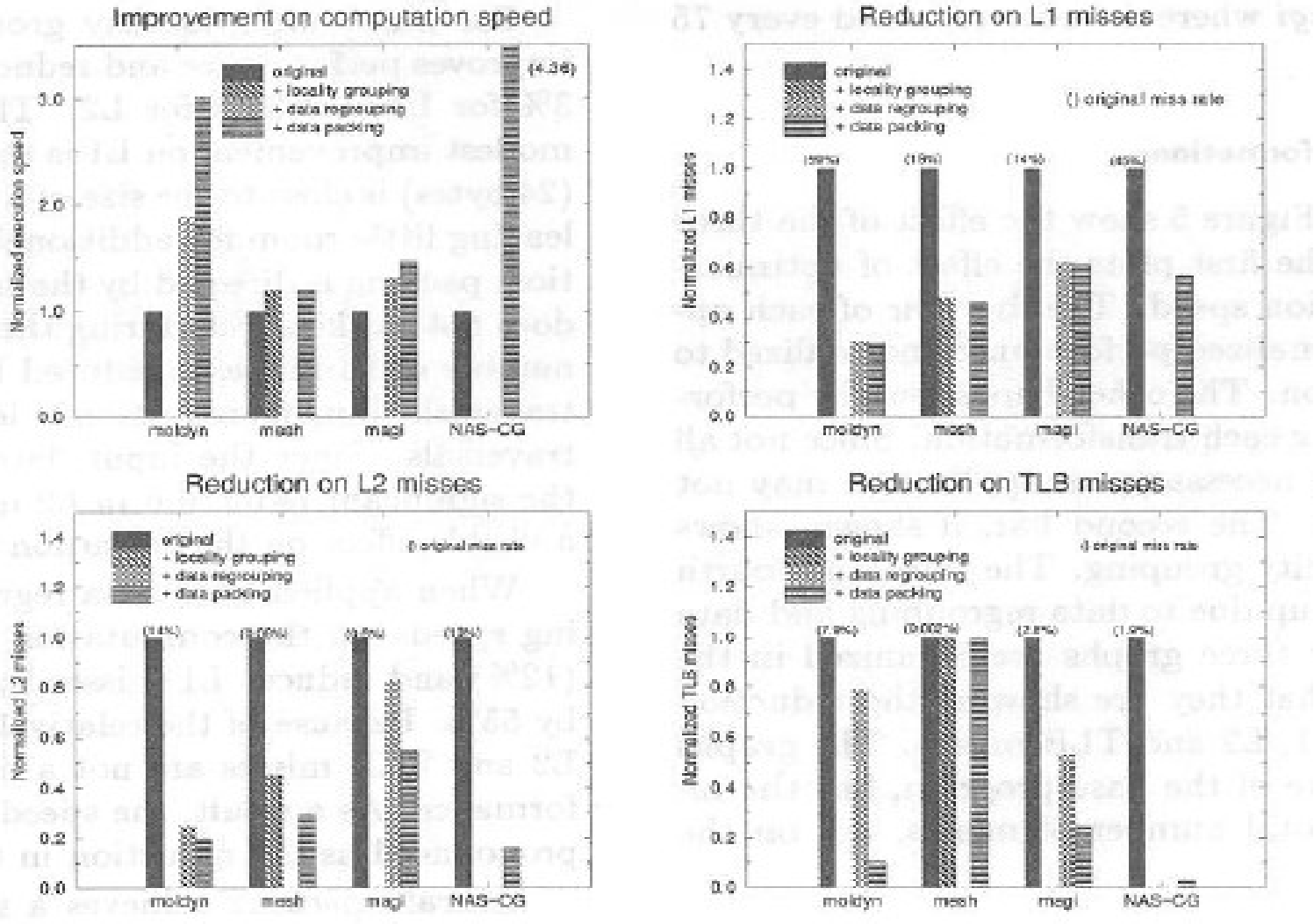


Figure 5: Effect of Transformations

Empirical Evaluation: Effects of Optimizations

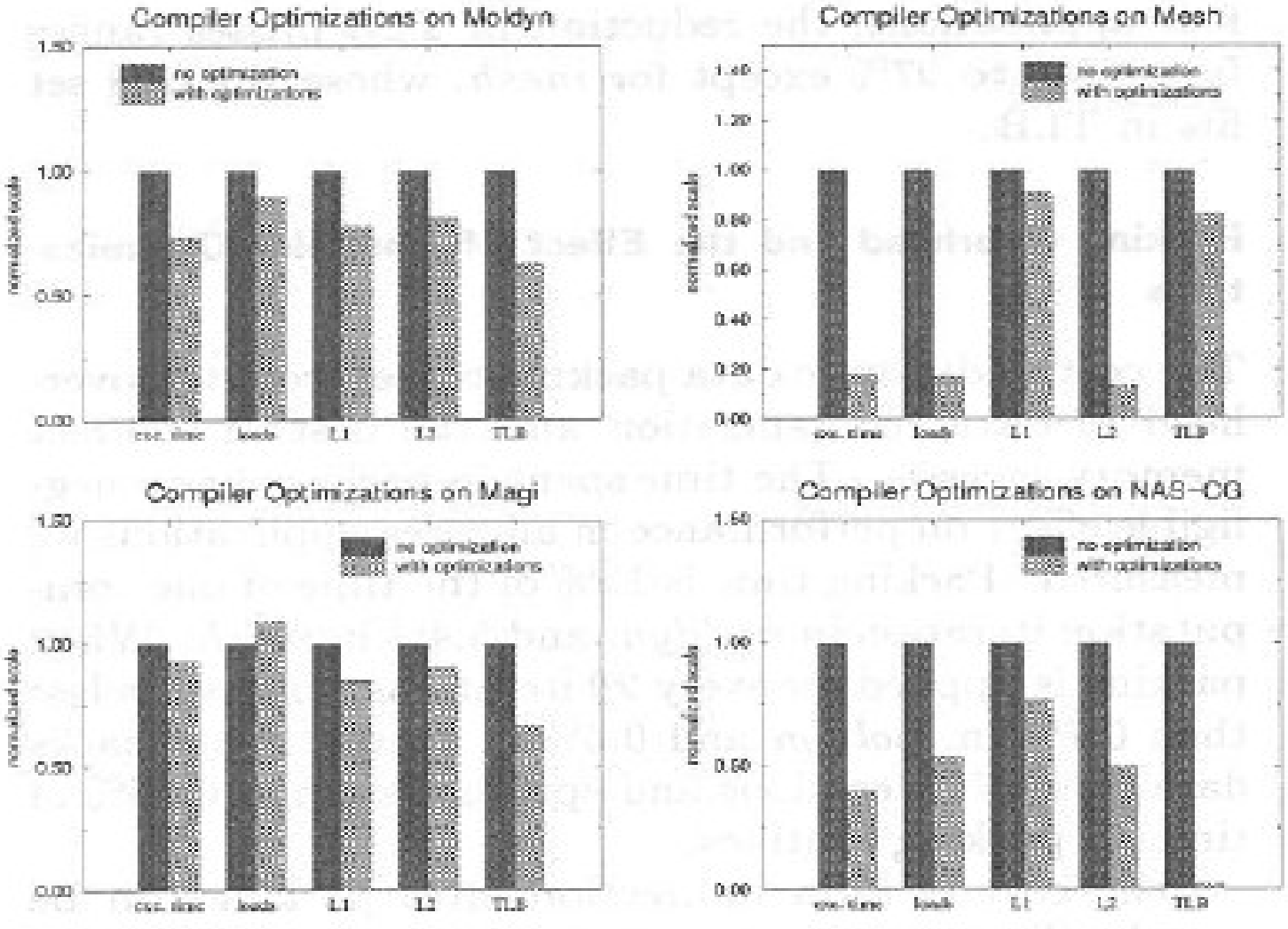
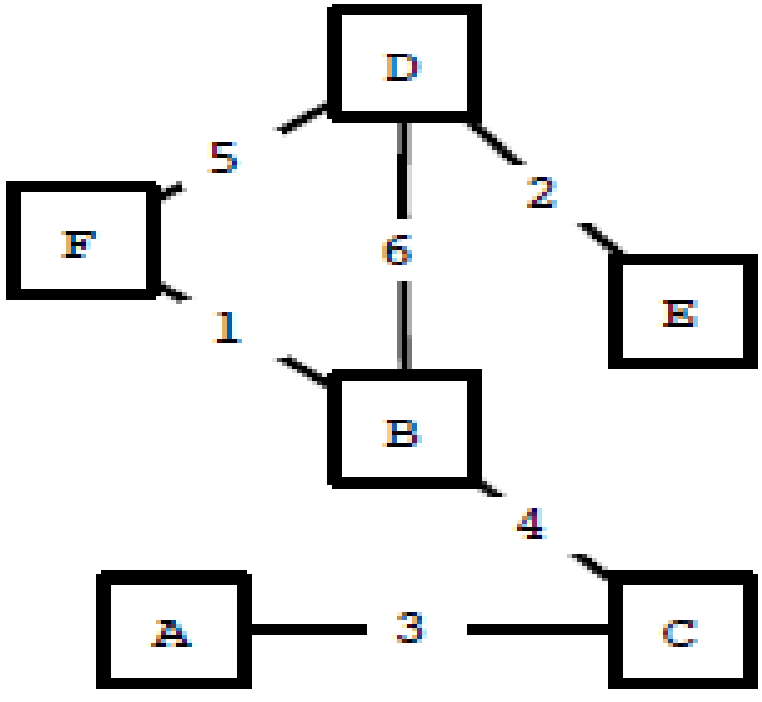


Figure 6: Effect of Compiler Optimizations

Spatial Locality Graph Models Data Reordering

- vertices correspond to data items, and
- an edge connect items accesses in the same iteration, and is annotated by the iteration number.



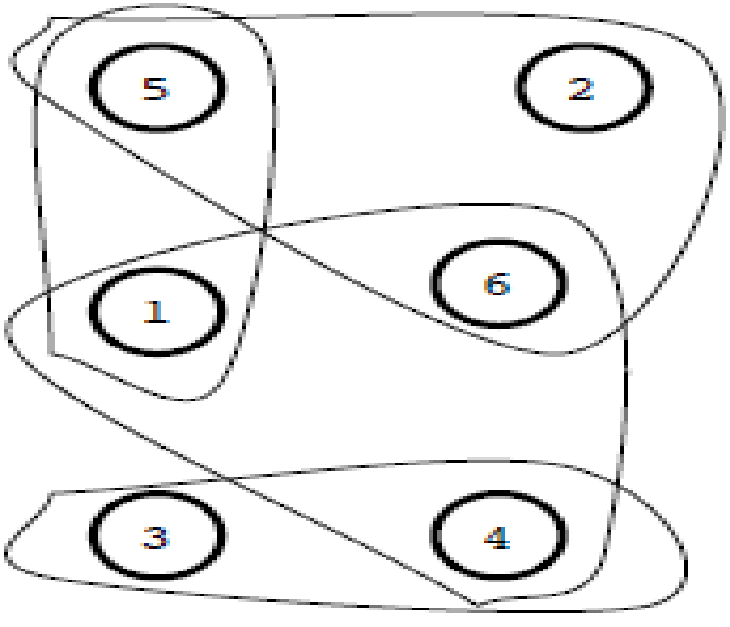
- G_{SL}: compute σ that minimizes $\sum_{(v,w)\in G_{SL}(E)} |\sigma(ind_v) - \sigma(ind_w)|$.

1 Consecutive Packing (CPACK): traverses the edges in the current iteration order and packs data on a first-come-first-served basis.

2 GPart: Heuristic that partitions the graph such that the nodes (data) of each partition fits into some level of cache, and orders the data consecutively (CPACK) inside each partition.

Temporal Locality HyperGraph (Iter Reordering)

- A HyperGraph $G_{TL}(V, E)$ is a generalization of a graph in which each hyperedge can involve more than 2 vertices.
- A vertex correspond to an iteration (number)
- A hyperedge is a set of vertices. Two or more vertices (iterations) belong to the same hyperedge if they access the same data item.



1. CPACKiter: visits the hyperedges in order and packs the iterations in each of these hyperedges on a first-come-first-served basis.
2. BFSiter: performs a BFS ordering of the vertixes of the hypergraph. Alg uses also $G_{SL}(V, E)$.
3. HPart: graph partitioning heuristics.

Temporal Locality HyperGraph (Iter Reordering)

```
Alg BFSiter(G_TL(V,E), G_SL(V,E)))
count = 0
ADD a vertex i∈ G_TL(V,E) TO iter-queue
DO {
  WHILE(iter-queue not empty) do {
    i = dequeue(iter-queue)
    put i next in iteration ordering
    count ++

    // for all hyperedges to which i belongs
    FOR EACH (v,w) = E_i ∈ G_SL(V,E))
      IF not visited, add v to data-queue & mark
      IF not visited, add w to data-queue
                                  and mark visited
    // Add all iters belonging to data hyperedges
    // to iter-queue
    WHILE (v = dequeue(data-queue))
      FOR EACH i in v, where v ∈ G_TL(E)
        IF not visited, add i to iter-queue & mark
  }
  if (count < n) add a non visited node to iter-queue
} WHILE (count < n)  // until all nodes were visited.
```

- Metric to minimize:
  $\sum_{e \in G_{TL}(E)} (\sum_{i_j, i_k \in e} |\delta(i_j) - \delta(i_k)|)$,
  where e is a hyperedge of the temporal locality hypergraph and δ gives the iteration new ordering.
- Span metric:
  $\sum_{e \in G_{TL}(E)} (max_{i \in e}(\delta(i)) - min_{i \in e}(\delta(i)))$
- Density metric:
  $\sum_{e \in G_{TL}(E)} (\frac{max_{i \in e}(\delta(i)) - min_{i \in e}(\delta(i))}{|e|})$
- Spatial Locality Hypergraph is the dual of the Temporal Locality Hypergraph, i.e., vertices are data items and an hyperedge is formed by the set of items accessed by an iteration.

Empirical Results: Weak Scaling

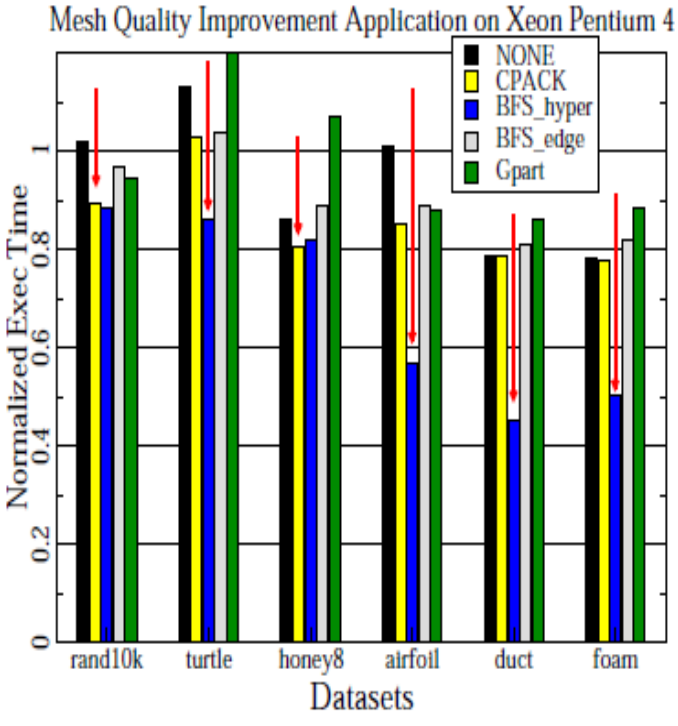


Figure 14: Results that compare various data-reordering heuristics applied to the mesh improvement application on the Xeon Pentium 4. Each bar represents the execution time for that dataset normalized to the execution time for the original ordering of that dataset. Each data reordering is followed by BFSIter for iteration reordering. The arrow indicates which data reordering results in the lowest spatial locality metric value for each dataset.

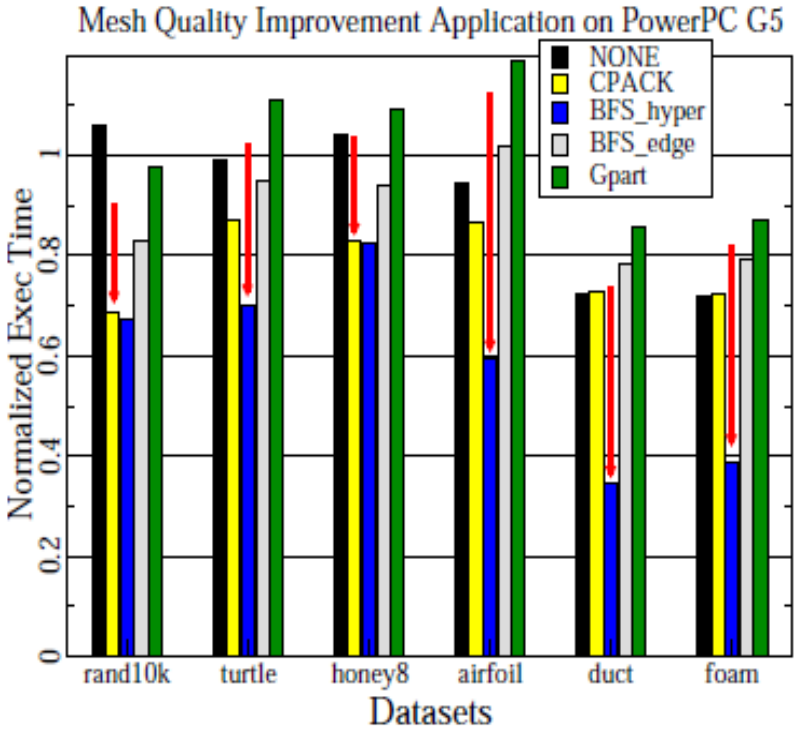


Figure 15: Results that compare various data-reordering heuristics applied to the mesh improvement application on the PowerPC G5. Each bar represents the execution time for that dataset normalized to the execution time for the original ordering of that dataset. Each data reordering is followed by BFSIter for iteration reordering. The arrow indicates which data reordering results in the lowest spatial locality metric value for each dataset.

Application: Parallelization of Irregular Arrays

Code Generation for Parallel Execution of a Class of Irregular Loops on Distributed Memory Systems, M. Ravishankar et. al., ICS 2013

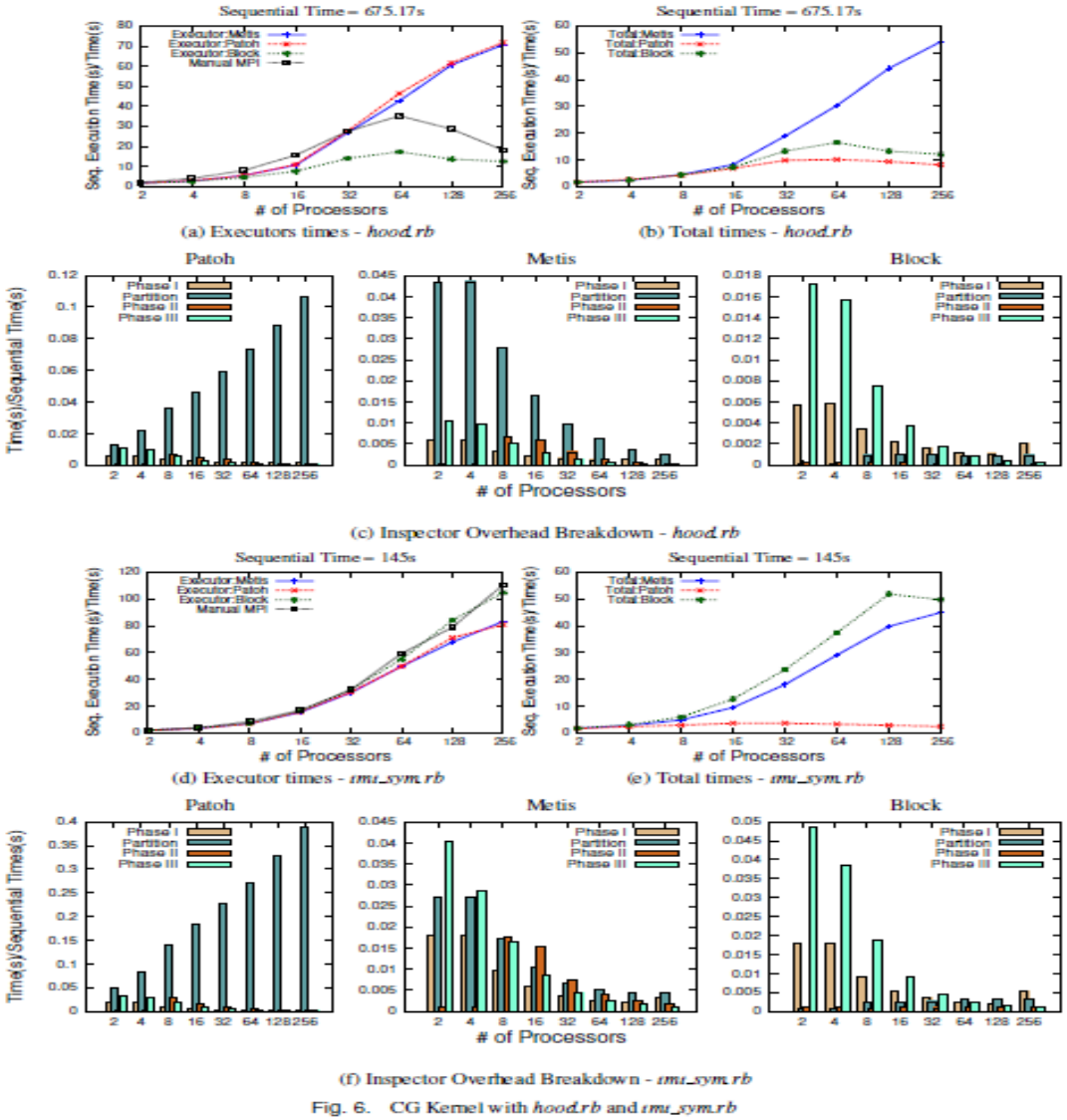
Sequential Conjugate Gradient Computation

```
while( !converged ) {
    //...Other computation not shown...
    //parallel, producer loop
    for( k = 0 ; k < n ; k++ )
        x[k] = ...;

    //...Other computation not shown...
    //parallel, consumer loop
    for( i = 0 ; i < n ; i++ )
        for( j = ia[i] ; j < ia[i+1] ; j++ ){
            xindex = col[j];
            y[i] += A[j]*x[xindex];
        }
    //...Other computation not shown...
}
```

- Generates automatically the inspector that determines which (indices of) elements of `x` (and `A`) are accessed in each outer iteration `i`, and builds the temporal locality hypergraph.
- Multi-Constraint Partitioning of the hypergraph (i) to achieve load balancing within each parallel loop and (ii) to minimize communication between the producer and consumer loops.

Application: Parallelization of Irregular Arrays



(a) Executors times - *hood.rb* (b) Total times - *hood.rb*

(c) Inspector Overhead Breakdown - *hood.rb*

(d) Executor times - *imi_sym.rb* (e) Total times - *imi_sym.rb*

(f) Inspector Overhead Breakdown - *imi_sym.rb*

Fig. 6. CG Kernel with *hood.rb* and *imi_sym.rb*

1. Tiling Affine Loop Nests: Optimising Communication & Load Balancing [Reddy and Bondhugula'14]

2. Iteration and Data Reordering for Loop Nests with Irregular Accesses [Ding and Kenedy'99], [Strout et. al. 2003,2004]
   - Data Reordering (Packing)
   - Iteration Reordering (Locality Grouping)
   - Generalization: Temporal & Spatial Locality HyperGraphs

3. Other Locality Optimizations: Parallel Tracing

Tracing Application: Copy (Garbage) Collector

A Localized Tracing Scheme Applied to Garbage Collection, Chicha and Watt, APLAS'06.
A New Approach to Parallelising Tracing Algorithms, Oancea, Mycroft and Watt, ISMM'09.

```
// Abstract Alg for Tracing:
1. mark and process any unmarked
          child of a marked node
2. repeat until no further marking
                      is possible

// One Sequential Implementation:
worklist.enqueue(root_items)
while worklist not empty
  item = worklist.dequeue()
  mark    item
  process item
  for each unmarked child of item
      worklist.enqueue(child)
```

Tracing Application: Copy (Garbage) Collector

A Localized Tracing Scheme Applied to Garbage Collection, Chicha and Watt, APLAS'06.
A New Approach to Parallelising Tracing Algorithms, Oancea, Mycroft and Watt, ISMM'09.

```
// Abstract Alg for Tracing:
1. mark and process any unmarked
          child of a marked node
2. repeat until no further marking
                      is possible

// One Sequential Implementation:
worklist.enqueue(root_items)
while worklist not empty
  item = worklist.dequeue()
  mark    item
  process item
  for each unmarked child of item
      worklist.enqueue(child)
```

- How to parallelize? Use several worklists instead of one. Worklist semantics:

1 Processor Centric: as many worklists as number of processors. A worklist holds items that are to be processed by the same processor.

2 Memory Centric: super-partition memory. A worklist is associated to a memory partition: it holds elements that belong to the same partition.

Semi Space Copy Collector

```
while(!queue.isEmpty()) {
  int ind = 0;
  Object from_child,to_child;
  Object to_obj = queue.dequeue();
  foreach (from_child in to_obj.fields()) {
    ind++;
    atomic{
      if( from_child.isForwarded() )
        continue;
      to_child = copy(from_child);
      setForwardingPtr(from_child,to_child);
    }
    to_obj.setField(to_child, ind-1);
    queue.enqueue(to_child);
} }
```

- Semi Space Collector partitions the memory into two halves: `from` and `out` space. When `from` space becomes full, the live objects are copied to the `out` space, and flips the role of the two spaces.

Semi Space Copy Collector

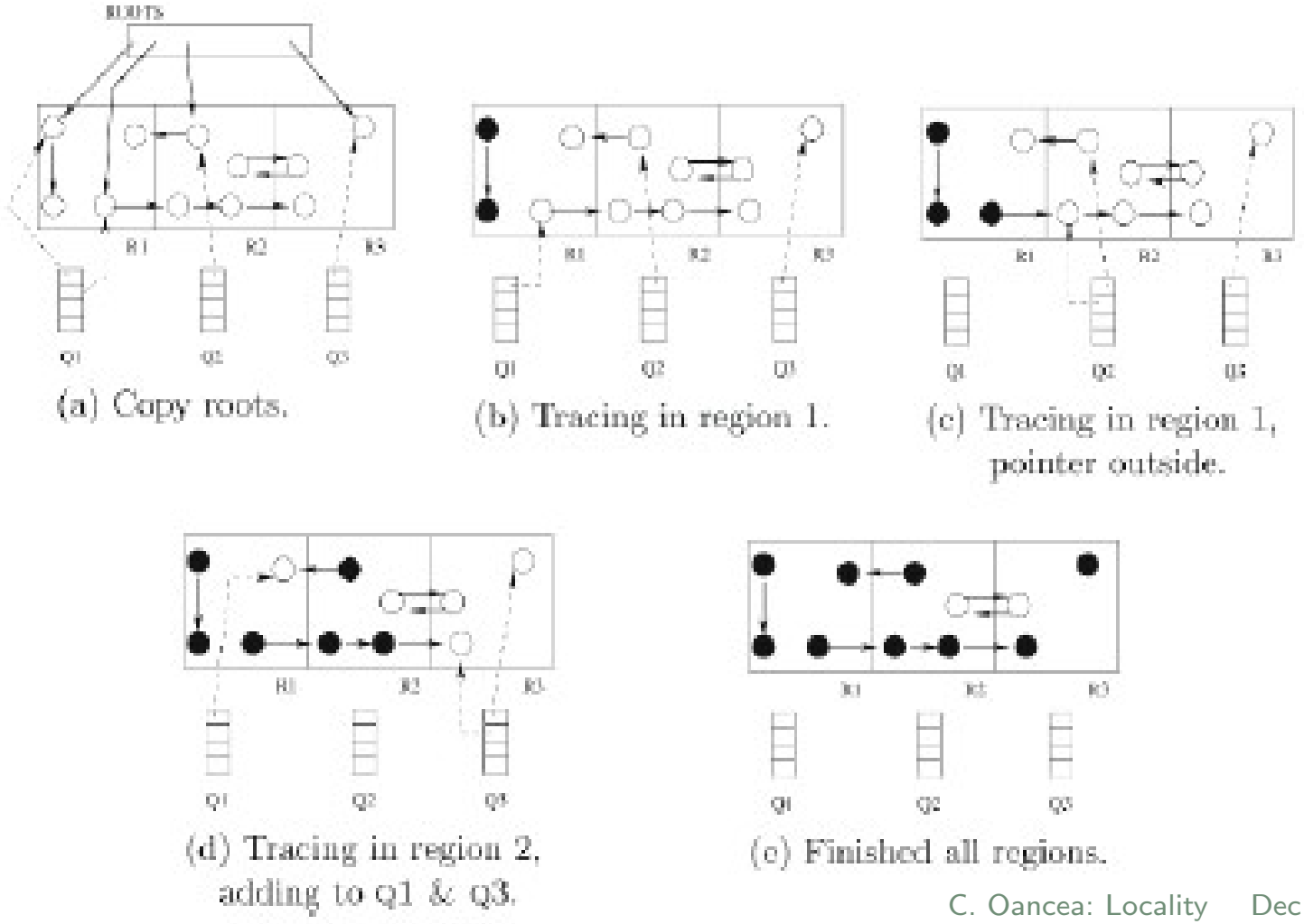
```
while(!queue.isEmpty()) {
  int ind = 0;
  Object from_child,to_child;
  Object to_obj = queue.dequeue();
  foreach (from_child in to_obj.fields()) {
    ind++;
    atomic{
      if( from_child.isForwarded() )
        continue;
      to_child = copy(from_child);
      setForwardingPtr(from_child,to_child);
    }
    to_obj.setField(to_child, ind-1);
    queue.enqueue(to_child);
} }
```

- Semi Space Collector partitions the memory into two halves: `from` and `out` space. When `from` space becomes full, the live objects are copied to the `out` space, and flips the role of the two spaces.
- `forwardingPtr` points to the `to` space object, and denotes marking.
- `to_obj.setField(to_child, ind-1);` sets field `ind-1` of `to_obj` to the copied child.

- Queue-access sync not a problem, e.g., double-ended queue data structure, that allows work stealing at minimal locking overhead.
- Problematic synchronization: the fine-grained, per object locking, without which an object can be copied to two to-space locations, with references split between the two.

Sequential Localized Tracing Scheme

Sequential, but uses several memory-centric worklist**s**. Reduces the working set and hence memory thrashing (TLB misses).



(a) Copy roots.

(b) Tracing in region 1.

(c) Tracing in region 1, pointer outside.

(d) Tracing in region 2, adding to q1 & q3.

(e) Finished all regions.

Parallel Localized Tracing Scheme

Memory-Centric Parallelization eliminates the problematic (fine-grained) synchronization overhead (when copying an object) because there is exactly one processor that owns a memory partition.