AUTOMATIC VECTORIZATION OF TREE TRAVERSALS

Youngjoon Jo, Michael Goldfarb and Milind Kulkarni

PURDUE
UNIVERSITY

PACT, Edinburgh, U.K.
September 11th, 2013

Commodity processors and SIMD

- Commodity processors support SIMD (Single Instruction Multiple Data) instructions
  - MMX (1996), SSE (1999), SSE2 (2001), SSE3 (2004), SSE4 (2006), AVX (2011), AVX2 (2013)
- SIMD width getting wider
  - AVX is 256bit
  - Upcoming AVX-512 to be 512bit (2015)
- Using SIMD is an excellent way to improve performance

SIMD works great for regular loops

```
for (int i = 0; i < 4; i++) {
  c[i] = a[i] + b[i];
}
```

SIMD works great for regular loops

```
for (int i = 0; i < 4; i++) {
  c[i] = a[i] + b[i];
}
```

↓

```
__m128 vec_a = _mm_load_ps(a);
__m128 vec_b = _mm_load_ps(b);
__m128 vec_c = _mm_add_ps(vec_a, vec_b);
_mm_store_ps(c, vec_c);
```

But not so well on irregular codes

```
void main() {
  Ray *rays[N] = // rays to trace
  Node *root = // root of tree
  for (int i = 0; i < N; i++) {
    recurse(rays[i], root);
  }
}

void recurse(Ray *r, Node *n) {
  if (truncate(r, n)) return;
  if (n->isLeaf()) {
    update(r, n);
  } else {
    recurse(r, n->left);
    recurse(r, n->right);
  }
}
```

But not so well on irregular codes

```
void main() {
  Ray *rays[N] = // rays to trace
  Node *root = // root of tree
  for (int i = 0; i < N; i++) {
    recurse(rays[i], root);
  }
}

void recurse(Ray *r, Node *n) {
  if (truncate(r, n)) return;
  if (n->isLeaf()) {
    update(r, n);
  } else {
    recurse(r, n->left);
    recurse(r, n->right);
  }
}
```

Automatic vectorization desired

```
void main() {
  Ray *rays[N] = // rays to trace
  Node *root = // root of tree
  for (int i = 0; i < N; i++) {
    re
  } else {
    recurse(r, n->left);
    recurse(r, n->right);
  }
}
```

Automatic vectorization techniques for irregular codes highly desired

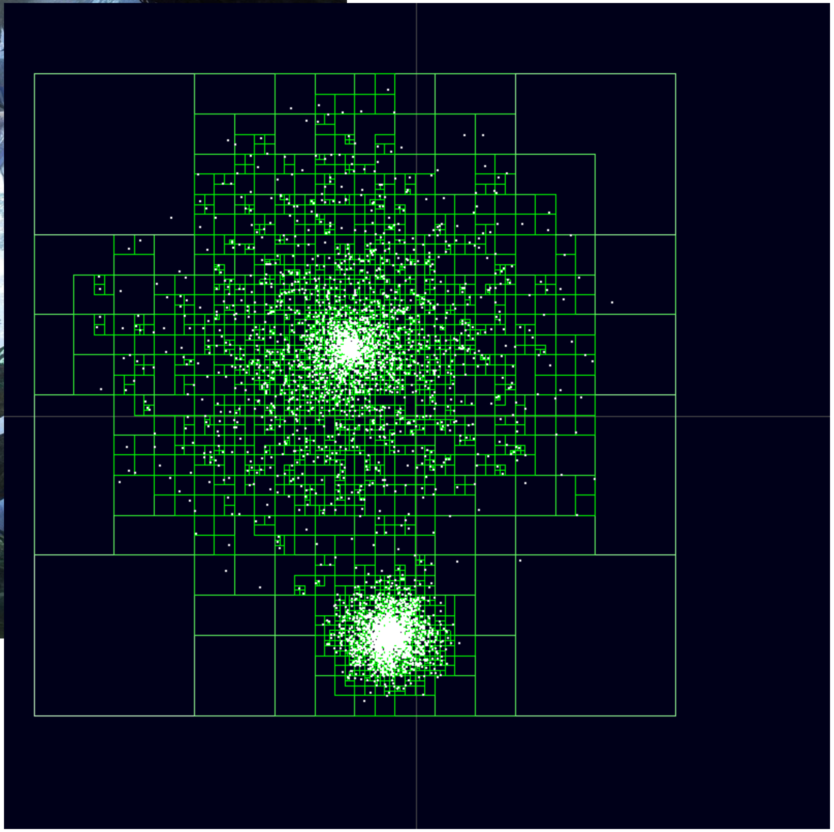
Automatic vectorization desired

```
void main() {
  Ray *rays[N] = // rays to trace
  Node *root = // root of tree
  for (int i = 0; i < N; i++) {
  } else {
    recurse(r, n->left);
    recurse(r, n->right);
  }
}
```

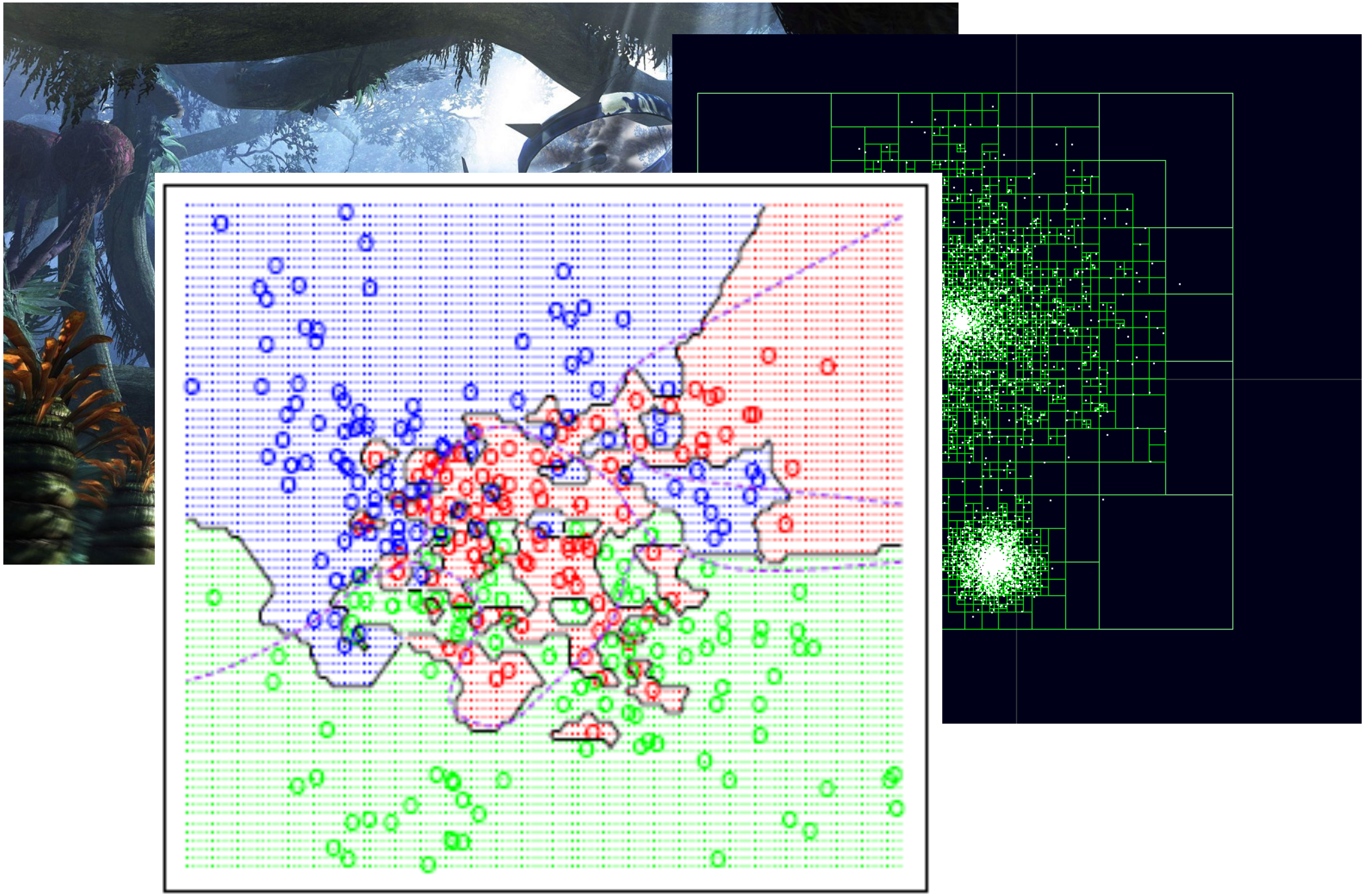
Automatic vectorization techniques for tree traversals highly desired

Tree codes are important

Tree codes are important



Tree codes are important



Tree vectorization challenges

- Non trivial to find vectorizable computation
- Difficult to keep vectorizable computation together

Previous tree vectorization work

- Non trivial to find vectorizable computation
  - Manually transform code to packetize traversals
  - Process multiple traversals in packet simultaneously
  - Wald et. al. [Computer Graphics Forum 2001]

- Difficult to keep vectorizable computation together

Previous tree vectorization work

In situations like physical simulation, collision detection or raytracing in scenes, where rays bounce into multiple directions (spherical or bumpmapped surfaces), coherent ray packets break down very quickly to single rays or do not exist at all. In the above mentioned tasks, packet oriented SIMD computations is much less useful.

Havel and Herout

[IEEE Transactions on Visualization and Computer Graphics 2010]

t to keep vectorizable computation together

Previous tree vectorization work

- Non trivial to find vectorizable computation
  - Manually transform code to packetize traversals
  - Process multiple points in packet simultaneously
  - Wald et. al. [Computer Graphics Forum 2001]

- Difficult to keep vectorizable computation together
  - Look to alternative sources of vectorization
  - Pixar [RT 2006]
    Dammertz et. al. [EGSR 2008]
    Kim et. al. [SIGMOD 2010]
    Chhugani et. al. [SC 2012]

Our previous tree locality work

- Point blocking
  Jo and Kulkarni [OOPSLA 2011]

- Traversal splicing
  Jo and Kulkarni [OOPSLA 2012]

Our solution

- Non trivial to find vectorizable computation
  - ~~Manually transform code to packetize traversals~~
  - Automatically packetize traversals with point blocking and a novel layout transformation
- Difficult to keep vectorizable computation together
  - ~~Look to alternative sources of vectorization~~
  - Exploit dynamic sorting of traversal splicing to dramatically enhance utilization

Contributions

- Show how tree traversal codes can be systematically transformed to
  - Expose SIMD opportunities
  - Enhance utilization
- Propose a novel layout transformation for efficient vectorization of tree codes
- Present a framework for automatically restructuring traversals and data layouts to enable vectorization

Contributions

- Show how tree traversal codes can be systematically transformed t
-
- and data layouts to enable vectorization

Spoiler alert!

SIMTree can deliver speedups of up to 6.59, and 2.78 on average

Outline

- Example & Abstract Model
- Point Blocking to Enable SIMD
- Traversal Splicing to Enhance Utilization
- Automatic Transformation
- Evaluation and Conclusion

Tree traversals

```
void main() {
  Ray *rays[N] = // rays to trace
  Node *root = // root of tree
  for (int i = 0; i < N; i++) {
    recurse(rays[i], root);
  }
}

void recurse(Ray *r, Node *n) {
  if (truncate(r, n)) return;
  if (n->isLeaf()) {
    update(r, n);
  } else {
    recurse(r, n->left);
    recurse(r, n->right);
  }
}
```

Tree traversals

```
void main() {
  Point *points[N] = // entities to traverse tree
  Node *root = // root of tree
  for (int i = 0; i < N; i++) {
    recurse(points[i], root);
  }
}

void recurse(Point *p, Node *n) {
  if (truncate(p, n)) return;
  if (n->isLeaf()) {
    update(p, n);
  } else {
    recurse(p, n->left);
    recurse(p, n->right);
  }
}
```

Tree traversals

```
void main() {
  Point *points[N] = // entities to traverse tree
  Node *root = // root of tree
  for (int i = 0; i < N; i++) {
    recurse(points[i], root);
  }
}

void recurse(Point *p, Node *n) {
  if (truncate(p, n)) return;
  if (n->isLeaf()) {
    update(p, n);
  } else {
    recurse(p, n->left);
    recurse(p, n->right);
  }
}
```

Tree traversals

```
void main() {
  Point *points[N] = // entities to traverse tree
  Node *root = // root of tree
  for (int i = 0; i < N; i++) {
    recurse(points[i], root);
  }
}

void recurse(Point *p, Node *n) {
  if (truncate(p, n)) return;
  if (n->isLeaf()) {
    update(p, n);
  } else {
    recurse(p, n->left);
    recurse(p, n->right);
  }
}
```

Tree traversals

```
void main() {
  Point *points[N] = // entities to traverse tree
  Node *root = // root of tree
  for (int i = 0; i < N; i++) {
    recurse(points[i], root);
  }
}

void recurse(Point *p, Node *n) {
  if (truncate(p, n)) return;
  if (n->isLeaf()) {
    update(p, n);
  } else {
    recurse(p, n->left);
    recurse(p, n->right);
  }
}
```

Tree traversals

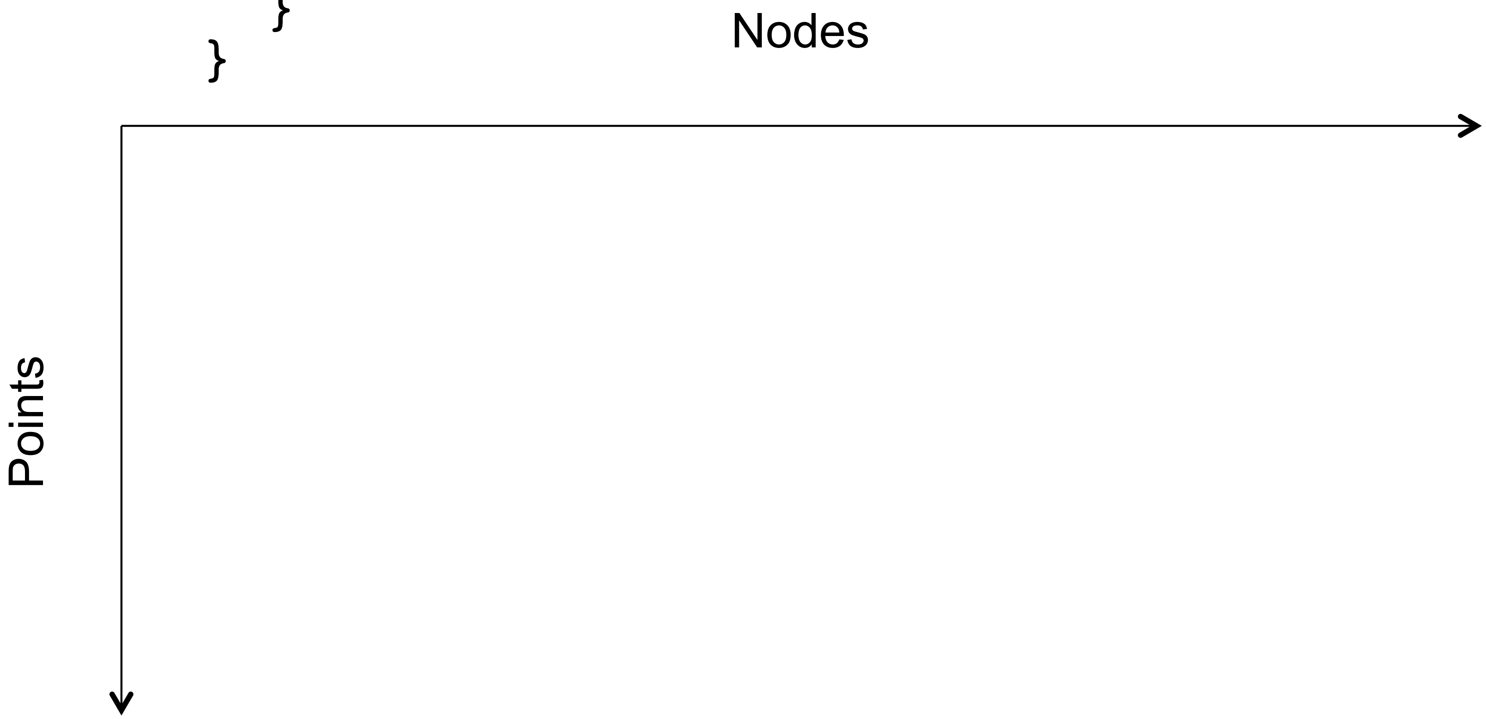
```
void main() {
  Point *points[N] = // entities to traverse tree
  Node *root = // root of tree
  for (int i = 0; i < N; i++) {
    recurse(points[i], root);
  }
}

void recurse(Point *p, Node *n) {
  if (truncate(p, n)) return;
  if (n->isLeaf()) {
    update(p, n);
  } else {
    recurse(p, n->left);
    recurse(p, n->right);
  }
}
```

An abstract model

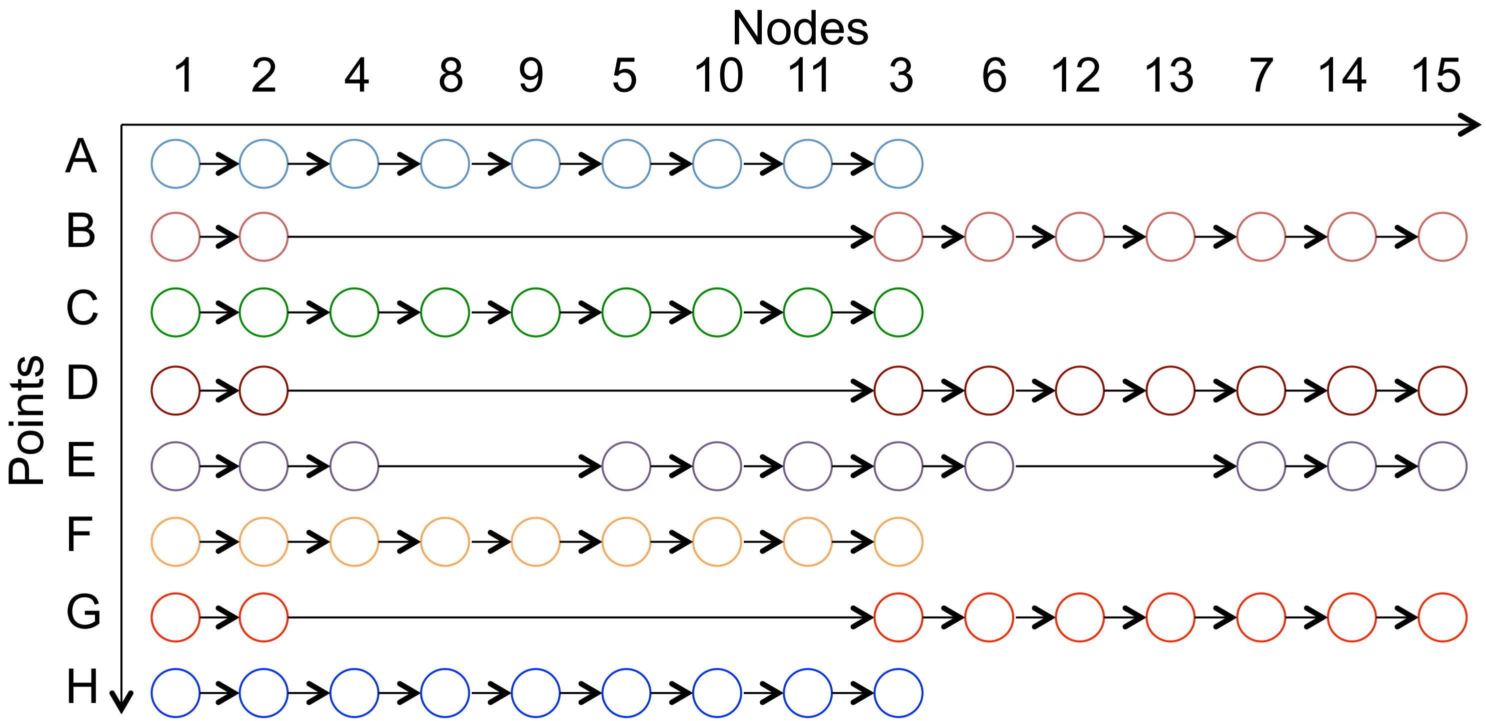
```
void main() {
  foreach(Point p : points) {
    foreach(Node n : p.oracleNodes()) {
      update(p, n);
    }
  }
}
```

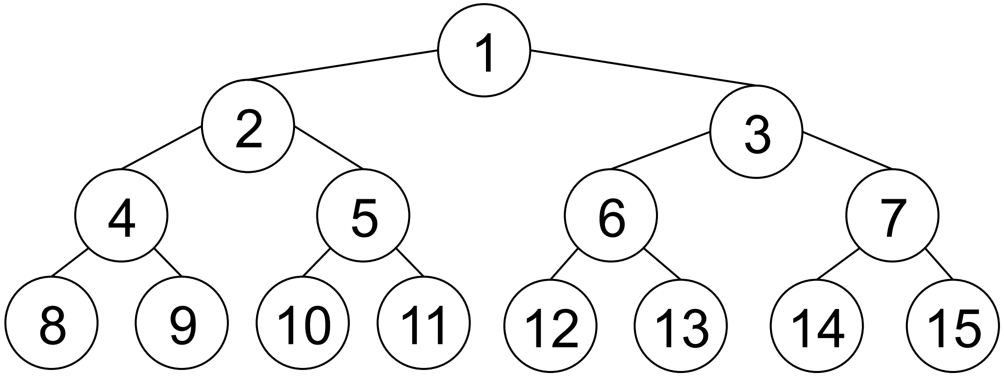
Iteration space of traversal

```
void main() {
  foreach(Point p : points) {
    foreach(Node n : p.oracleNodes()) {
      update(p, n);
    }
  }
}
```

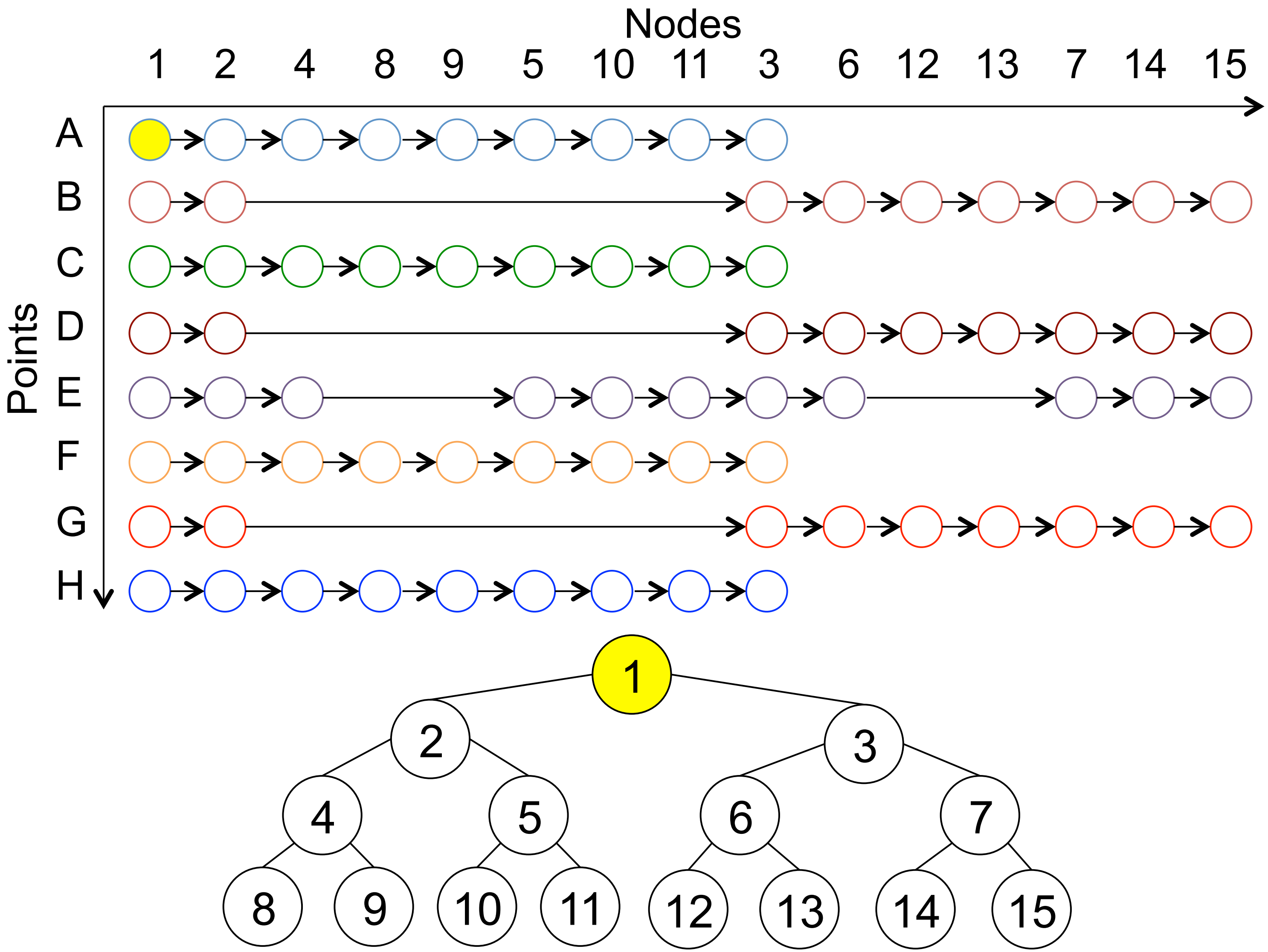


Iteration space of traversal

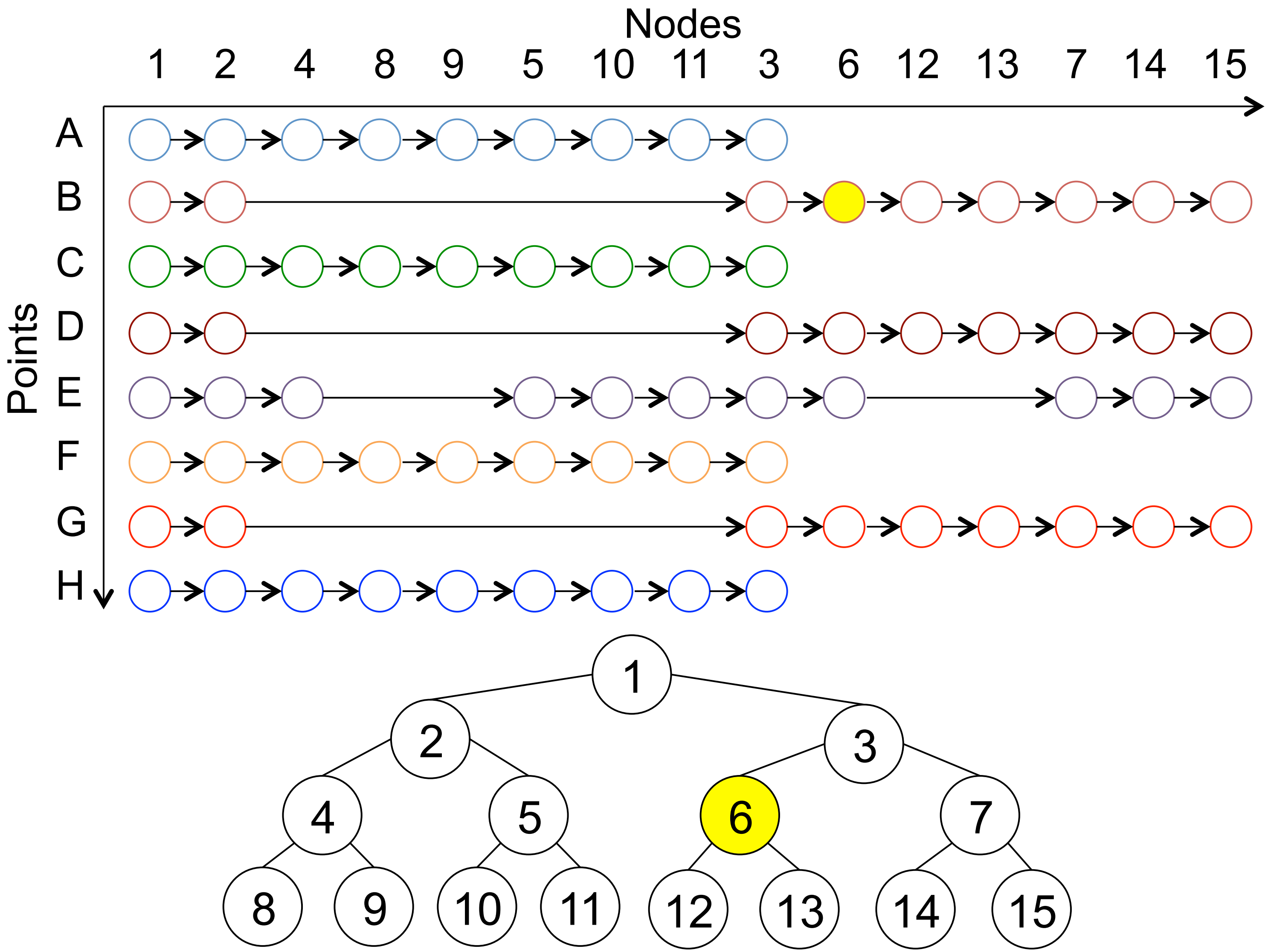




Iteration space of traversal



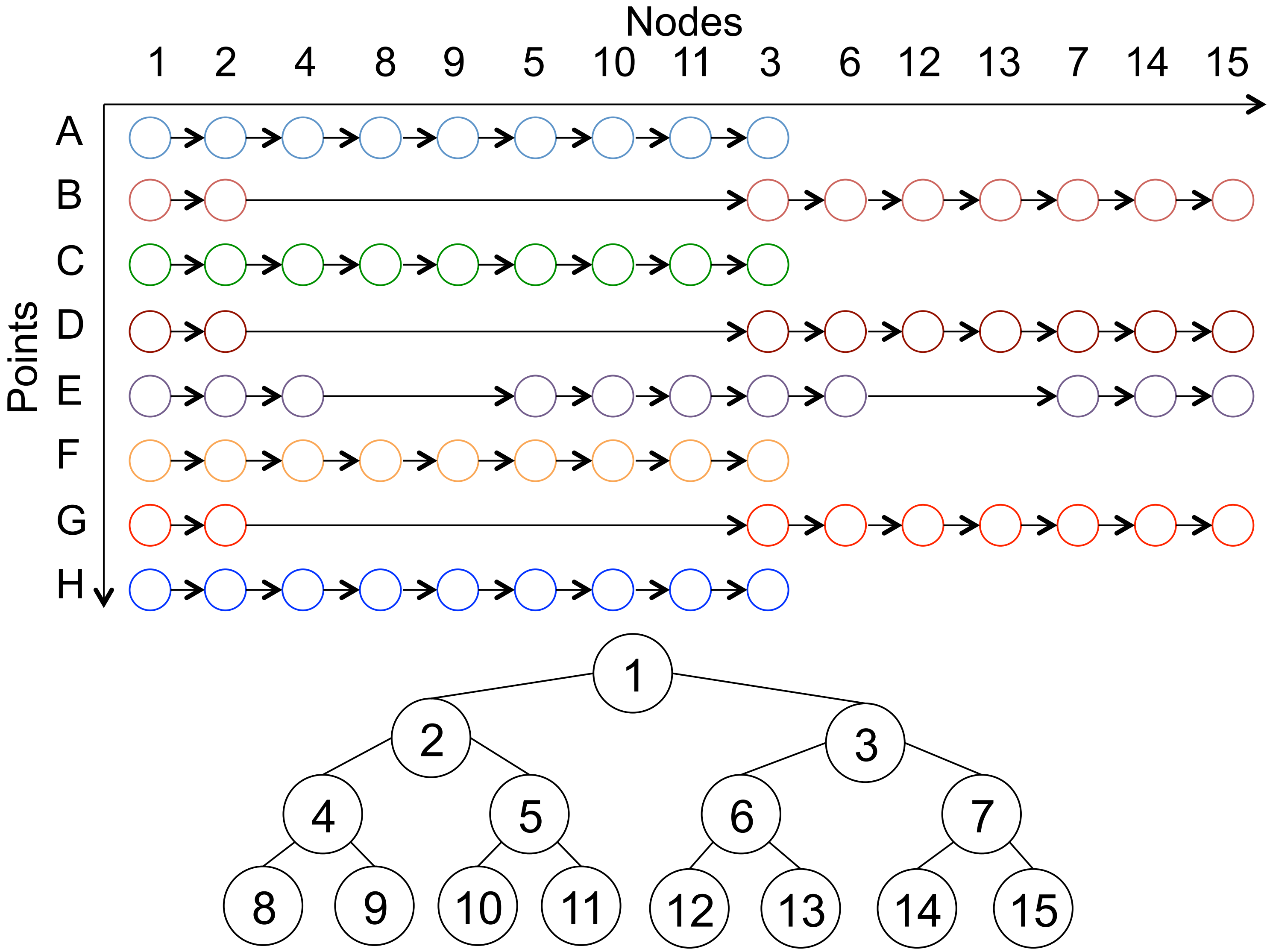
How to vectorize?



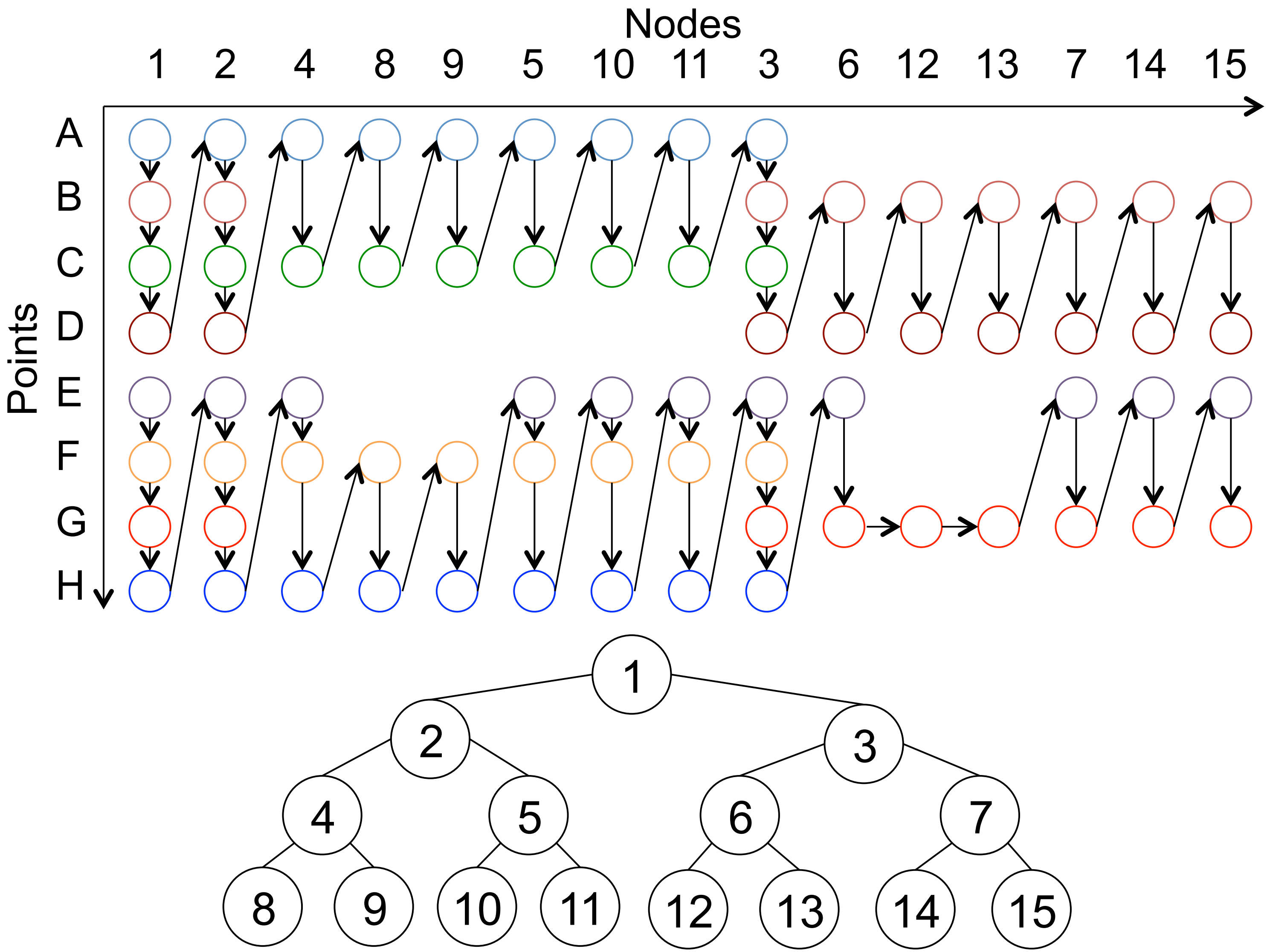
Outline

- Example & Abstract Model
- Point Blocking to Enable SIMD
- Traversal Splicing to Enhance Utilization
- Automatic Transformation
- Evaluation and Conclusion

Point blocking [OOPSLA 2011]



Point blocking [OOPSLA 2011]



Point blocked code

```
void recurse(Point *p, Node *n) {
  if (truncate(p, n)) return;
  if (n->isLeaf()) {
    update(p, n);
  } else {
    recurse(p, n->left);
    recurse(p, n->right);
  }
}
```

Point blocked code

```
void recurse(Block *block, Node *n) {
  if (truncate(p, n)) return;
  if (n->isLeaf()) {
    update(p, n);
  } else {
    recurse(p, n->left);
    recurse(p, n->right);
  }
}
```

Point blocked code

```
void recurse(Block *block, Node *n) {
  if (truncate(p, n)) return;
  if (n->isLeaf()) {
    update(p, n);
  } else {
    recurse(p, n->left);
    recurse(p, n->right);
  }
}
```

Function body

Point blocked code

Loop over points in block

```
void recurse(Block *block, Node *n) {
  for (int i = 0; i = block->size; i++) {
    Point *p = block->p[i];
    if (truncate(p, n)) continue;
    if (n->isLeaf()) {
      update(p, n);
    } else {
      recurse(p, n->left);
      recurse(p, n->right);
    }
  }
}
```

Function body

Point blocked code

Loop over points in block

```
void recurse(Block *block, Node *n) {
  for (int i = 0; i = block->size; i++) {
    Point *p = block->p[i];
    if (truncate(p, n)) continue;
    if (n->isLeaf()) {
      update(p, n);
    } else {
      recurse(p, n->left);
      recurse(p, n->right);
    }
  }
}
```

Function body

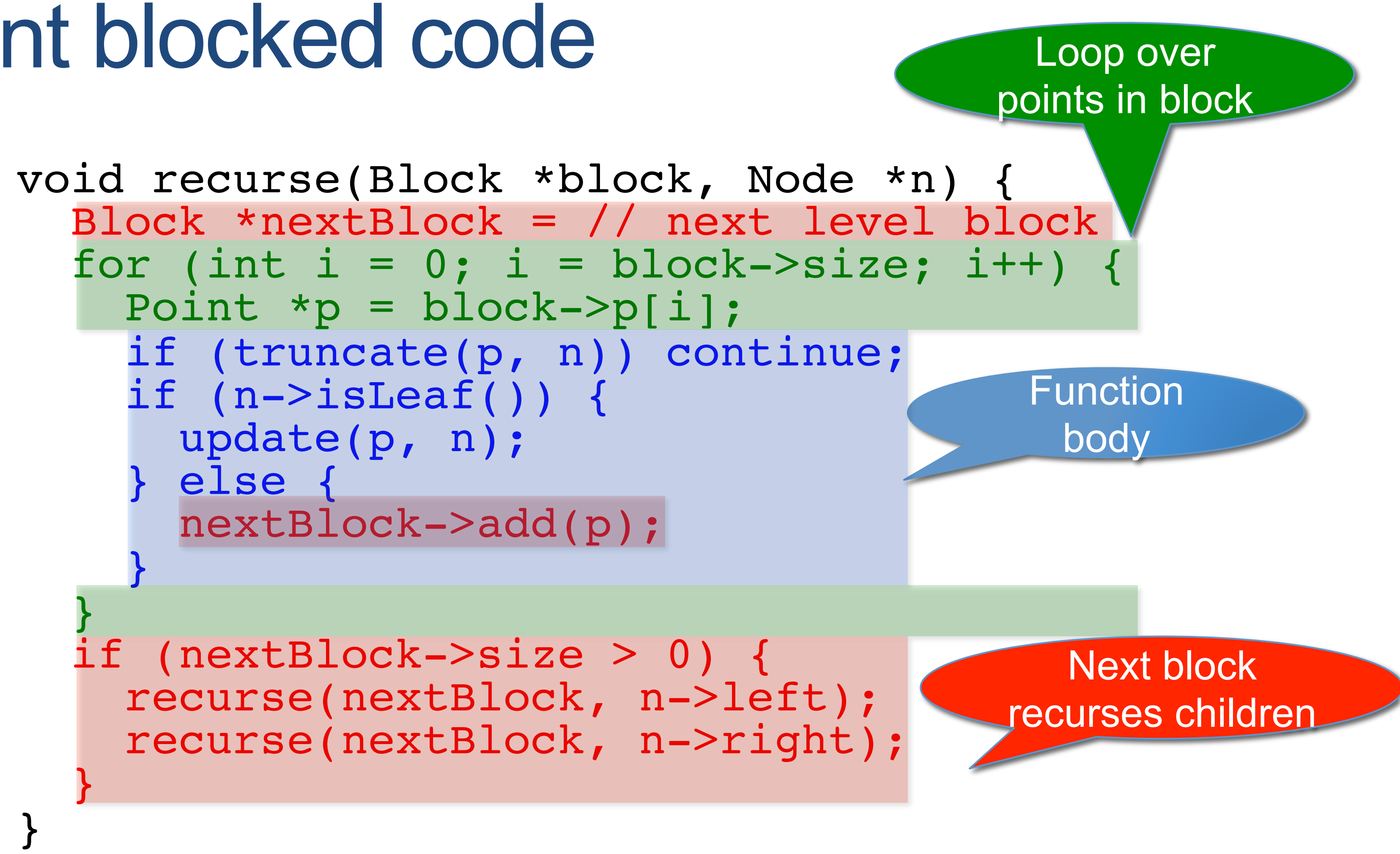
Point blocked code

Loop over points in block

```
void recurse(Block *block, Node *n) {
  Block *nextBlock = // next level block
  for (int i = 0; i = block->size; i++) {
    Point *p = block->p[i];
    if (truncate(p, n)) continue;
    if (n->isLeaf()) {
      update(p, n);
    } else {
      nextBlock->add(p);
    }
  }
}
```

Function body

Point blocked code



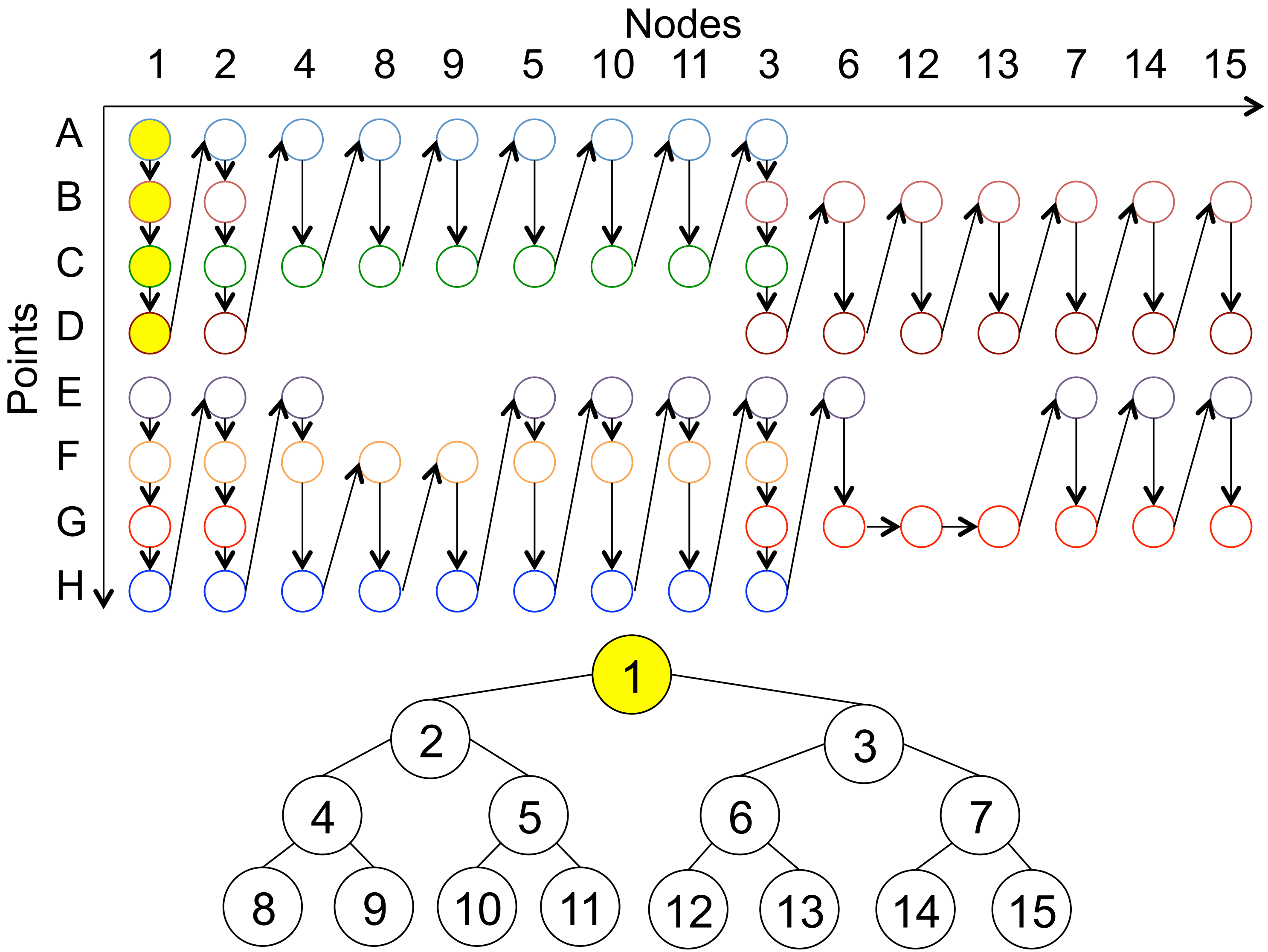
```
void recurse(Block *block, Node *n) {
  Block *nextBlock = // next level block
  for (int i = 0; i = block->size; i++) {
    Point *p = block->p[i];
    if (truncate(p, n)) continue;
    if (n->isLeaf()) {
      update(p, n);
    } else {
      nextBlock->add(p);
    }
  }
  if (nextBlock->size > 0) {
    recurse(nextBlock, n->left);
    recurse(nextBlock, n->right);
  }
}
```

Loop over points in block

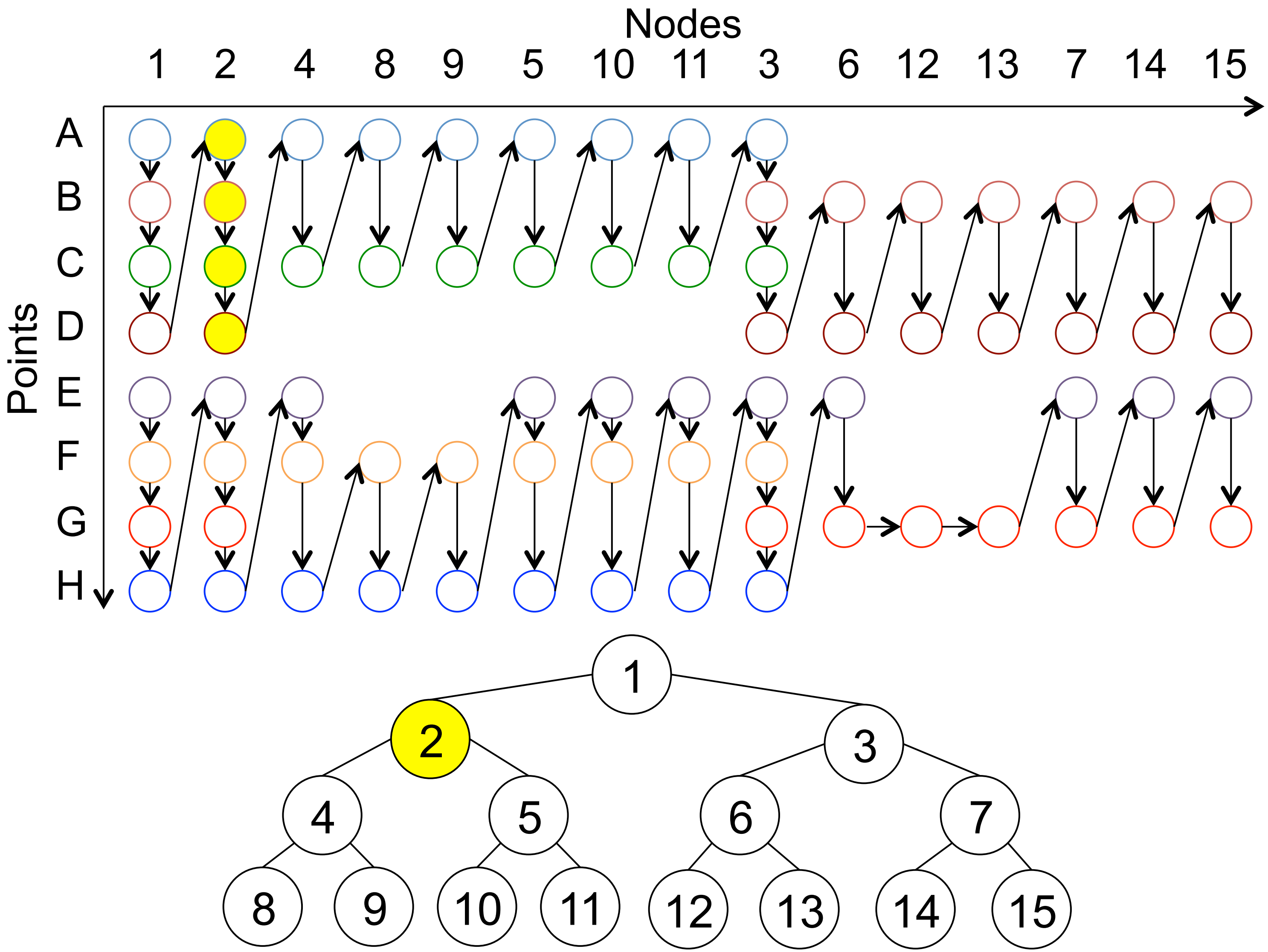
Function body

Next block recurses children

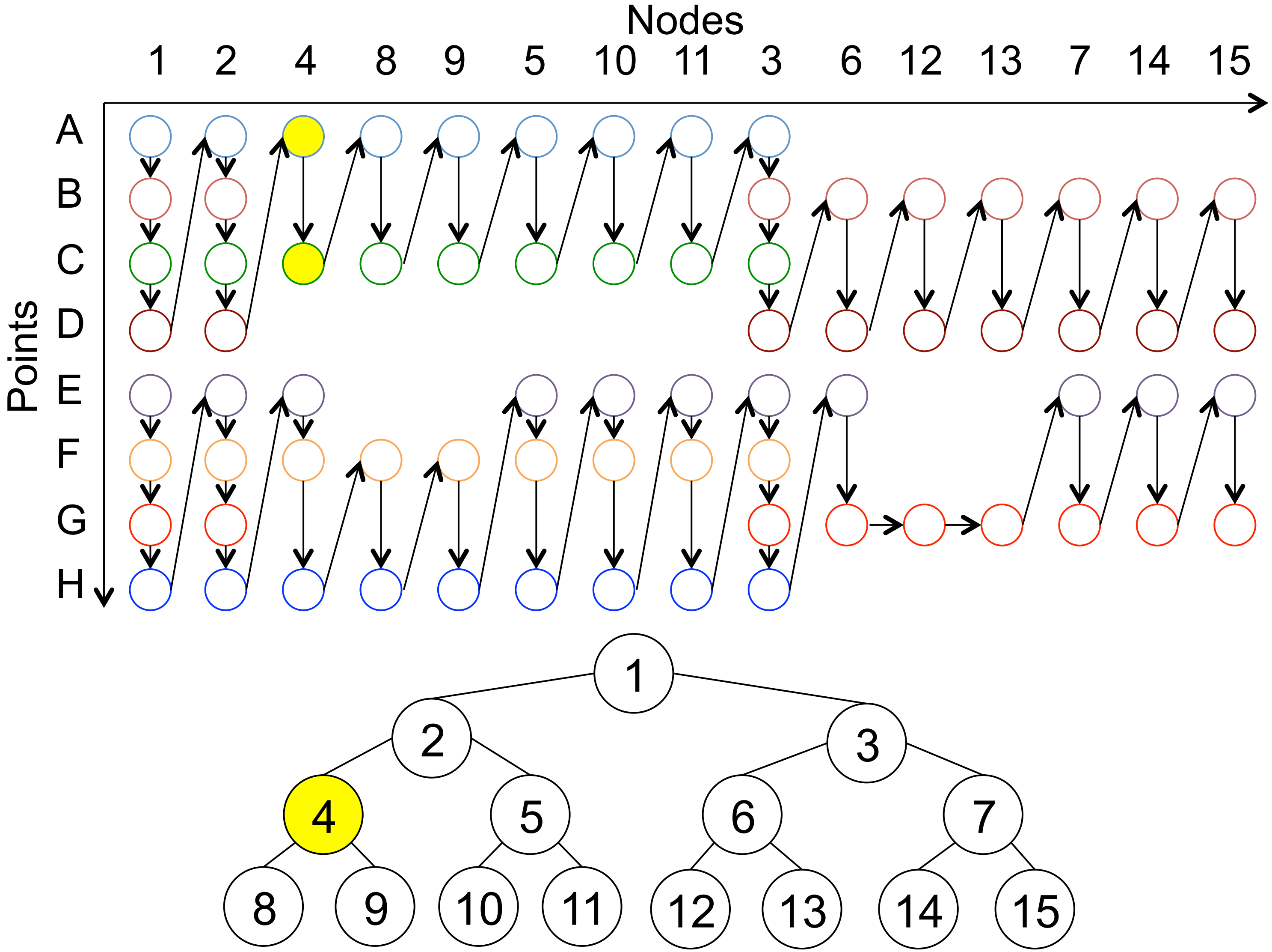
Point blocking [OOPSLA 2011]



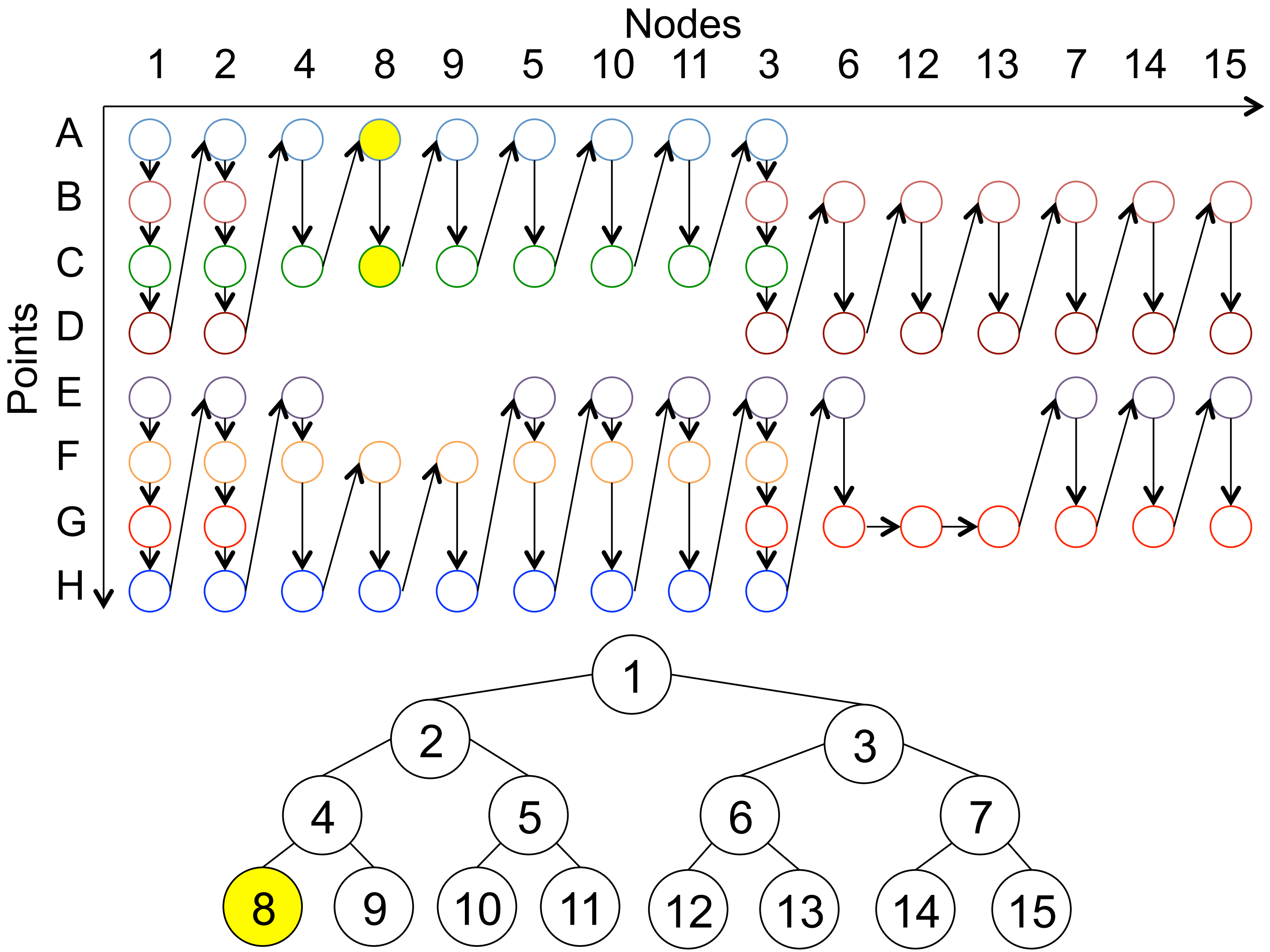
Point blocking [OOPSLA 2011]



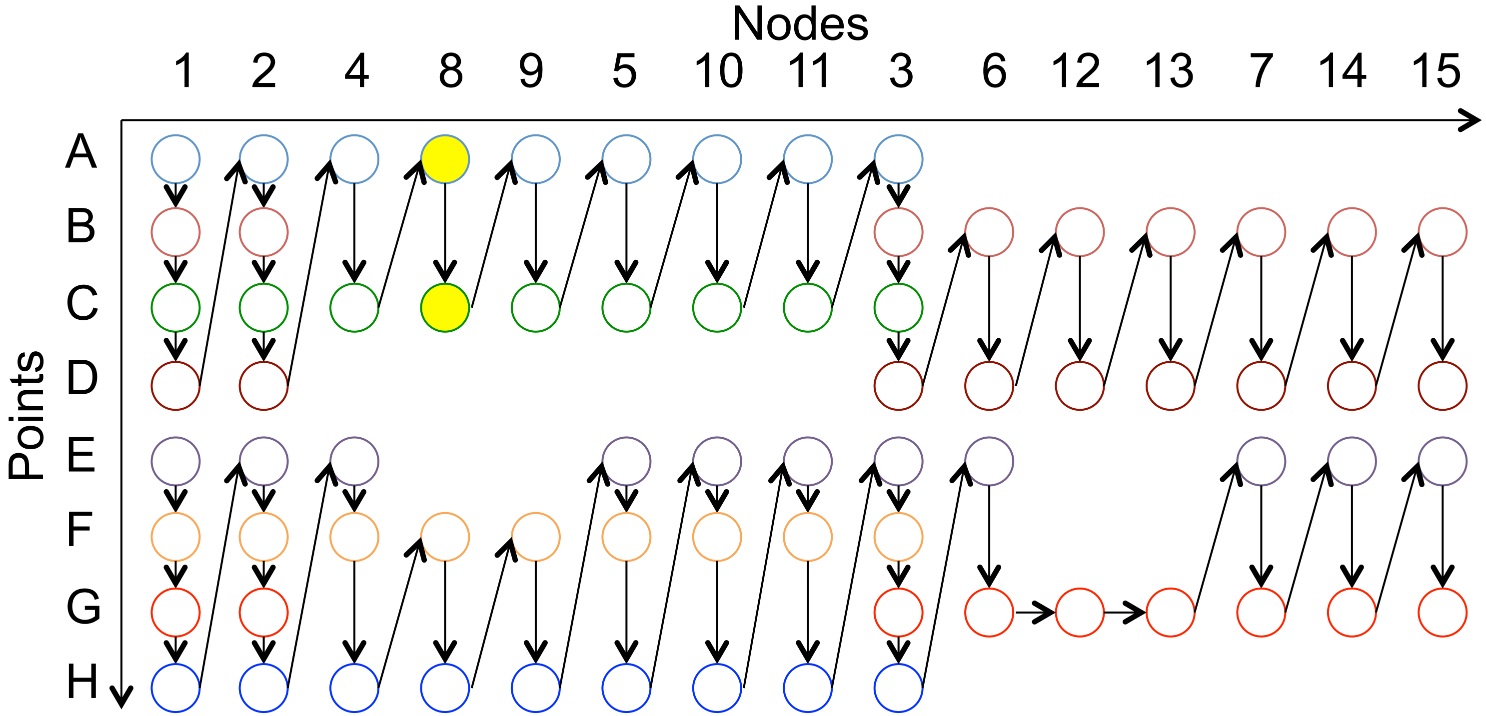
Point blocking [OOPSLA 2011]

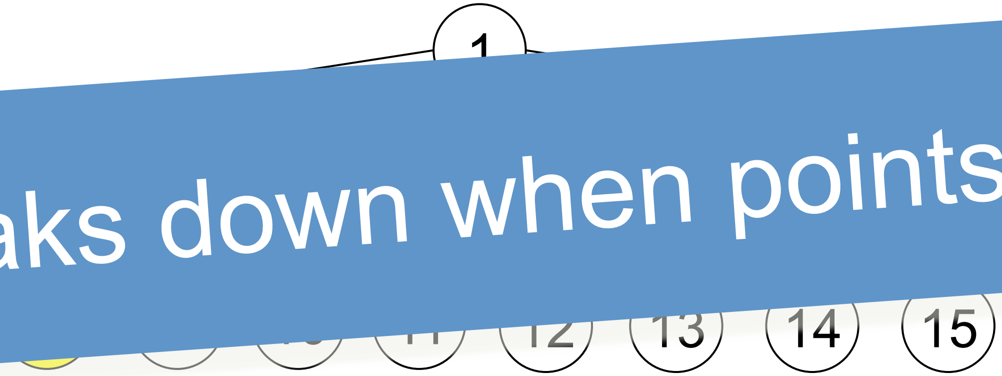


Analogous to packet SIMD



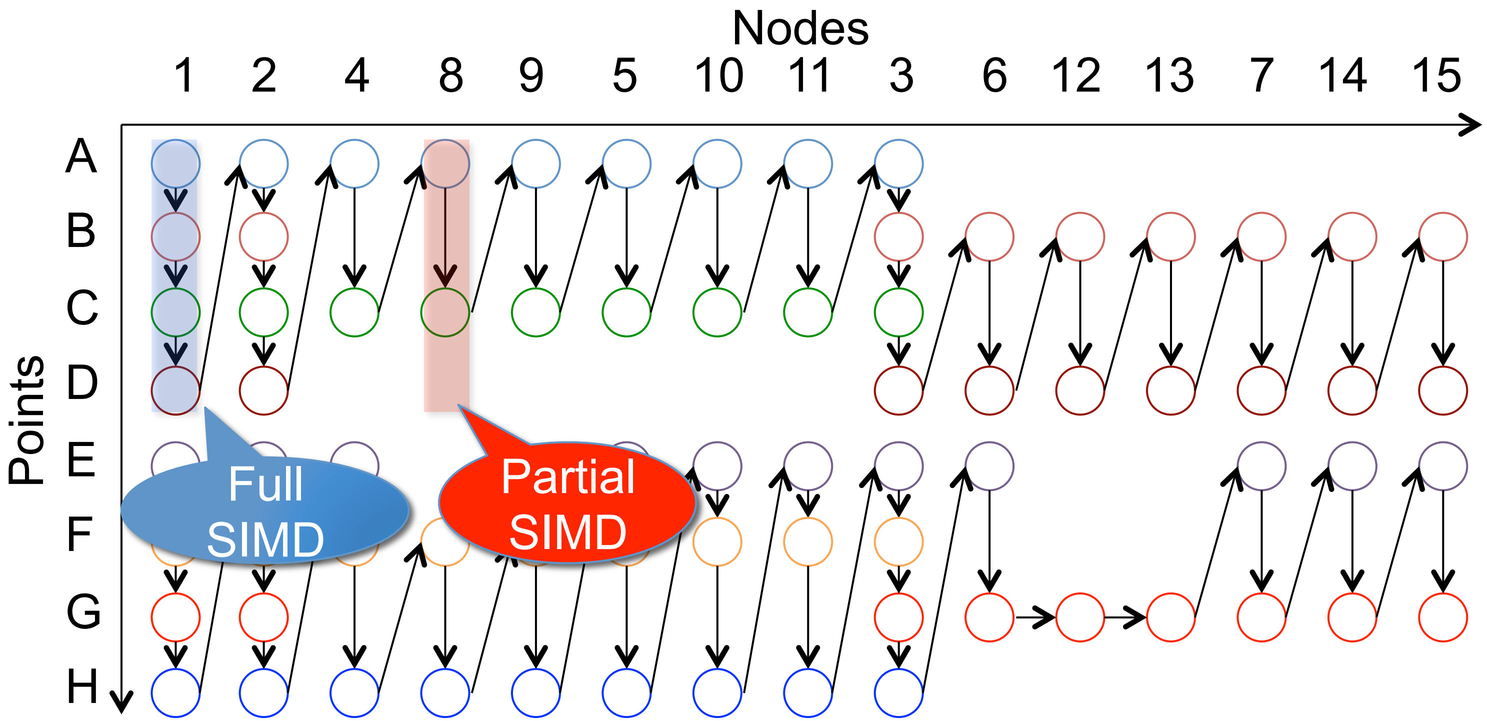
Analogous to packet SIMD

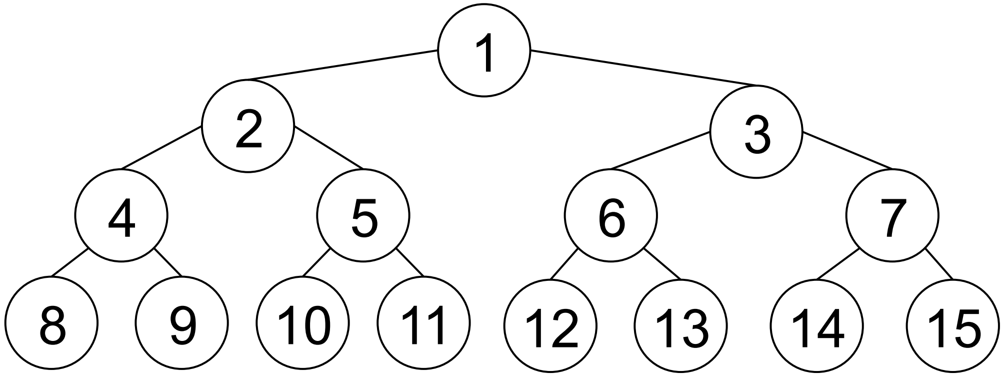




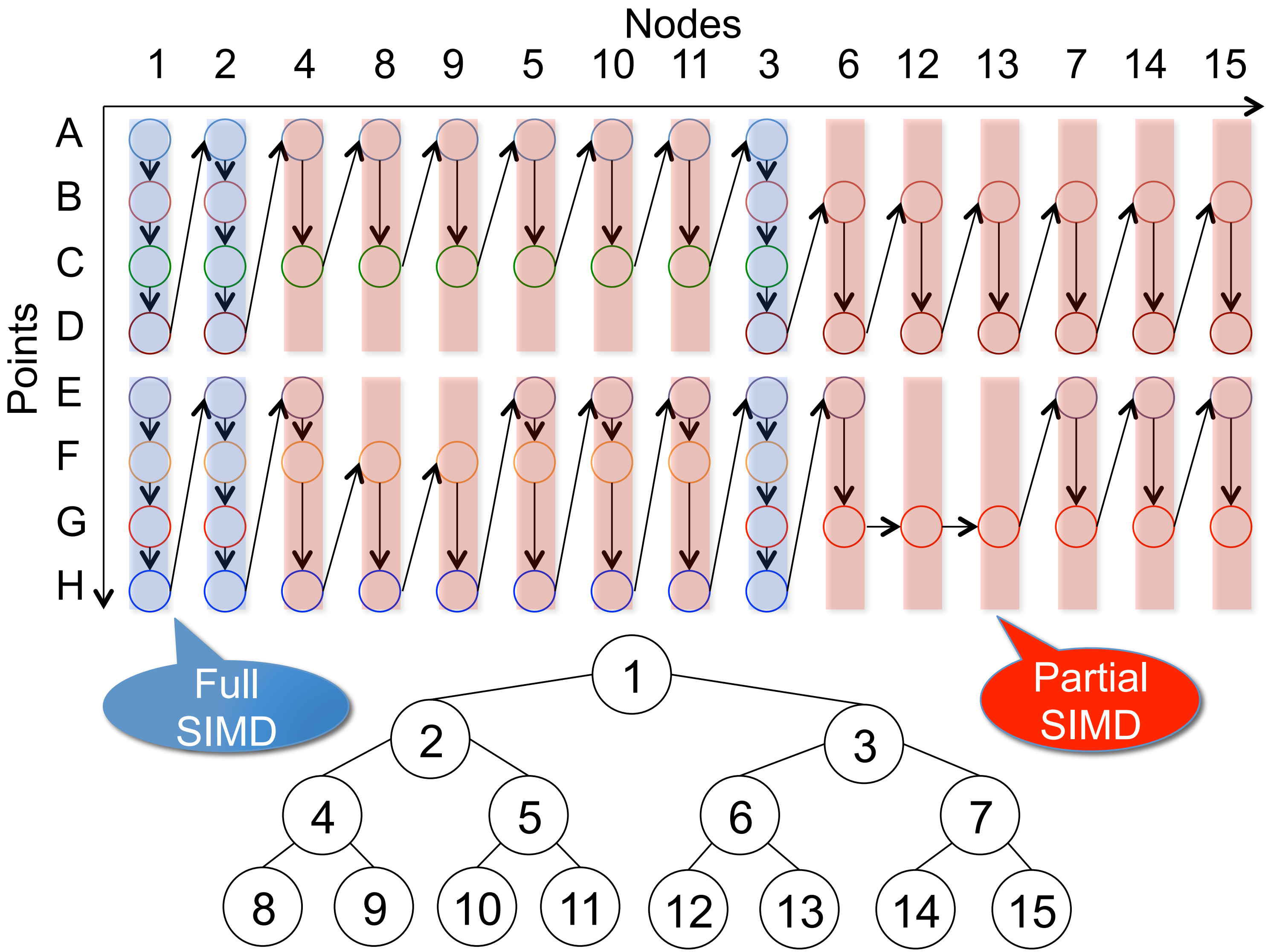
Breaks down when points diverge

Packet SIMD has poor utilization

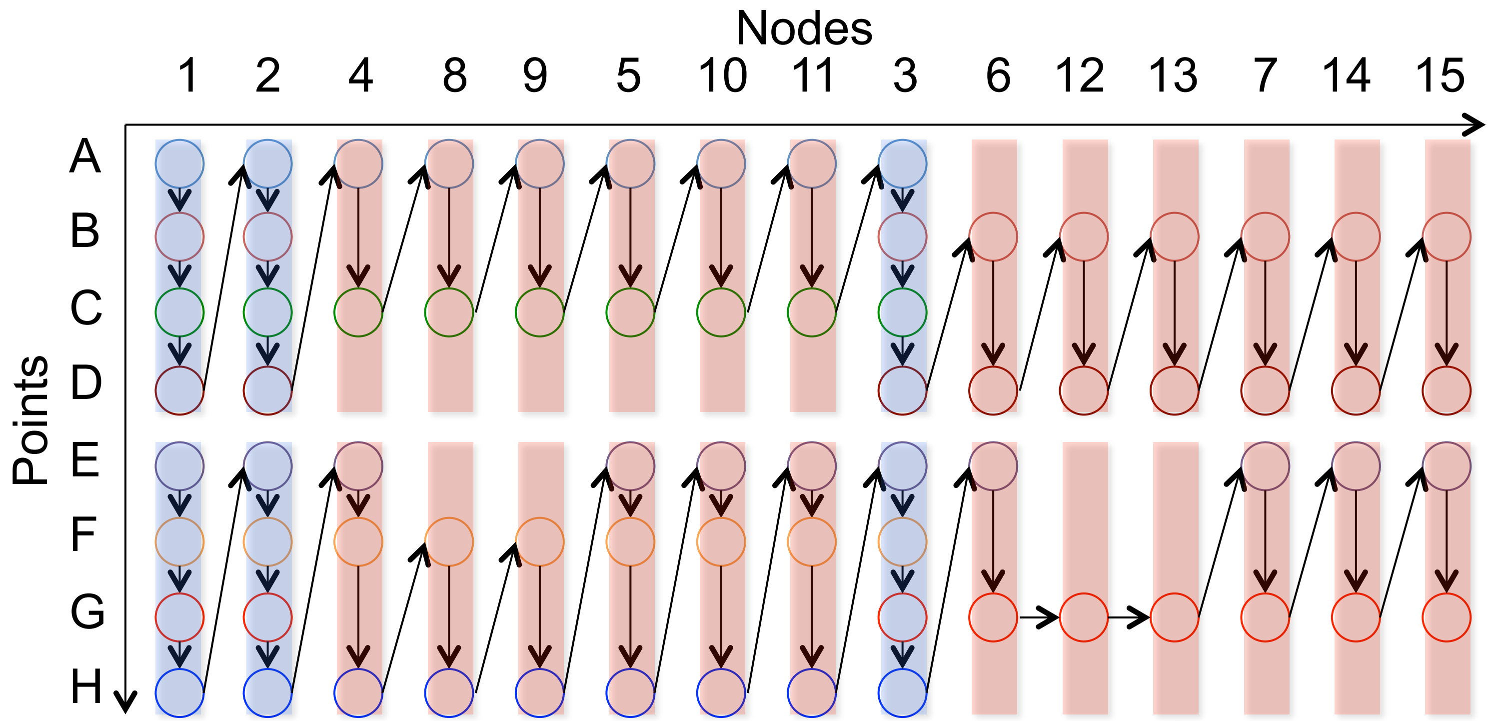




Packet SIMD has poor utilization



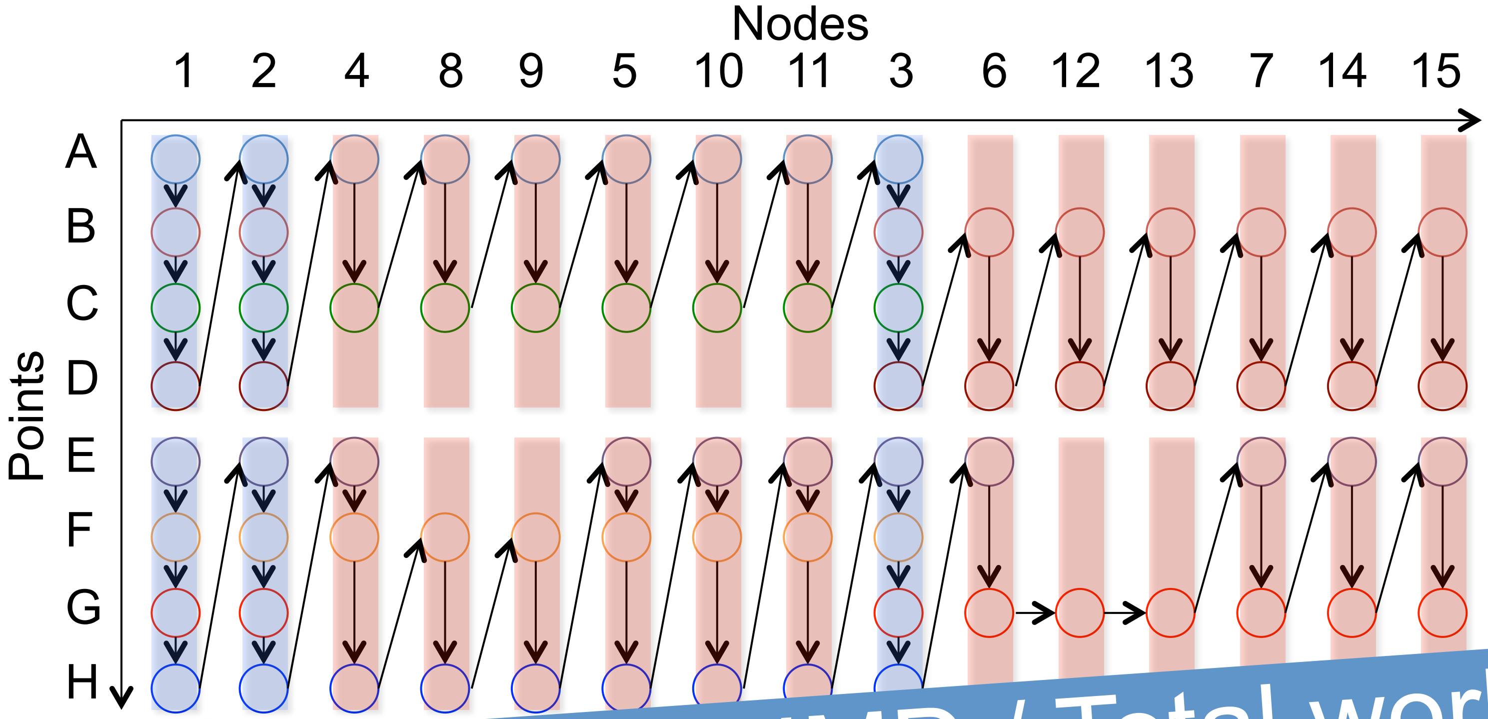
SIMD utilization



Full

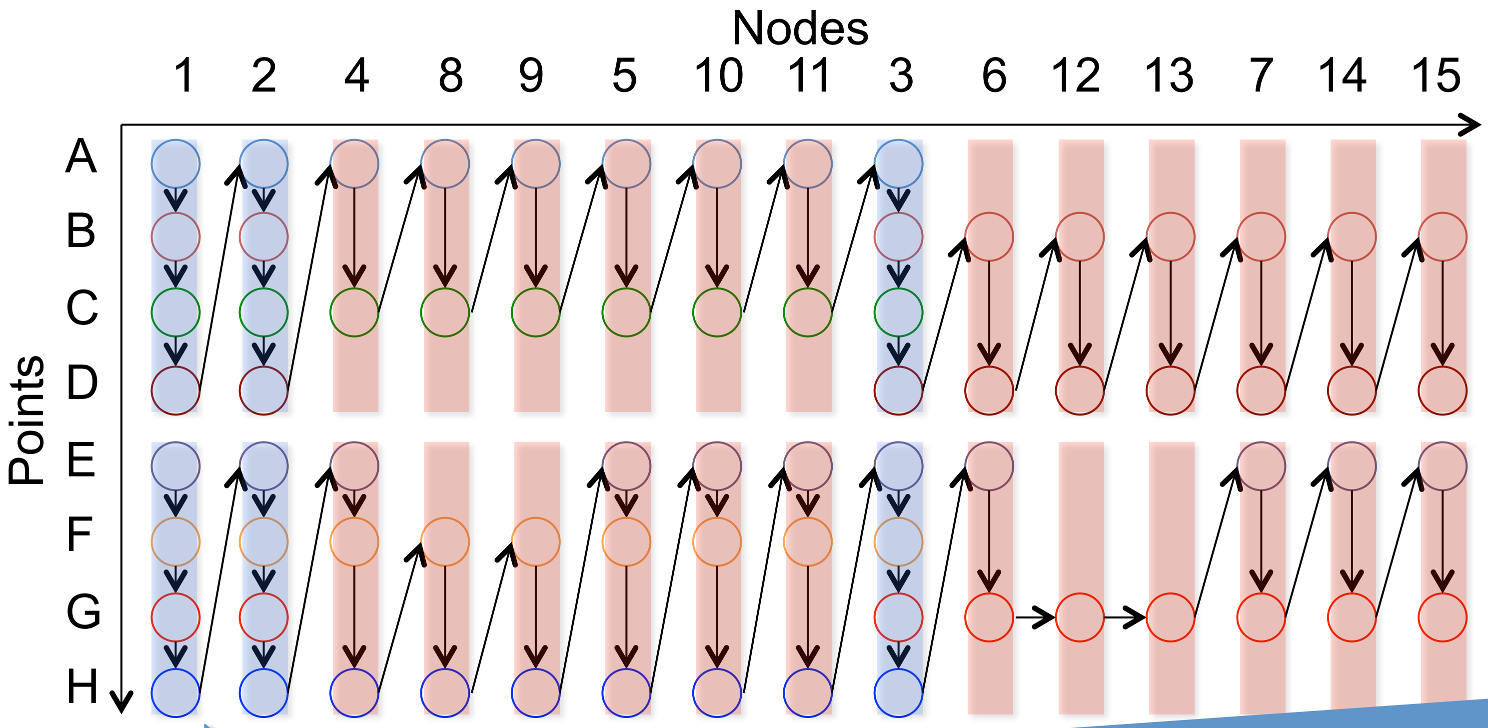
SIMD utilization
= Work in full SIMD / Total work

SIMD utilization



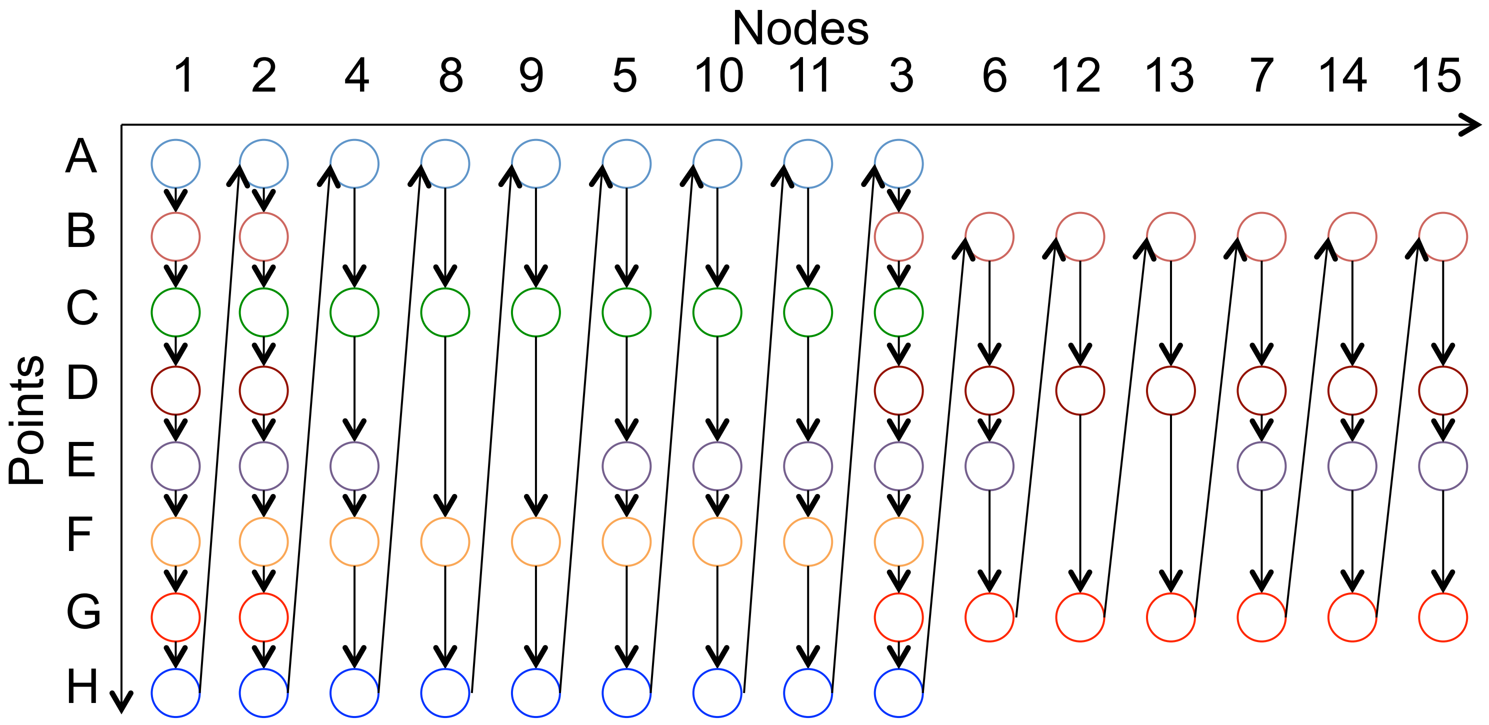
Work in full SIMD / Total work
= Circles in blue / Total circles
= 24 / 74 = 0.32

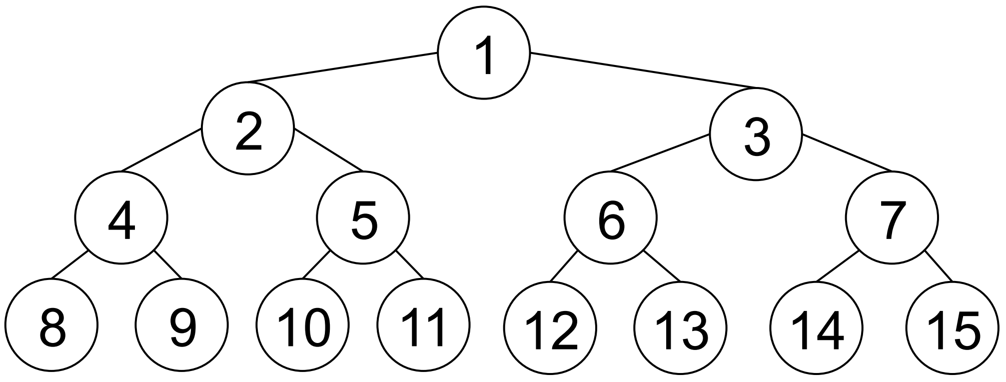
Use larger block size



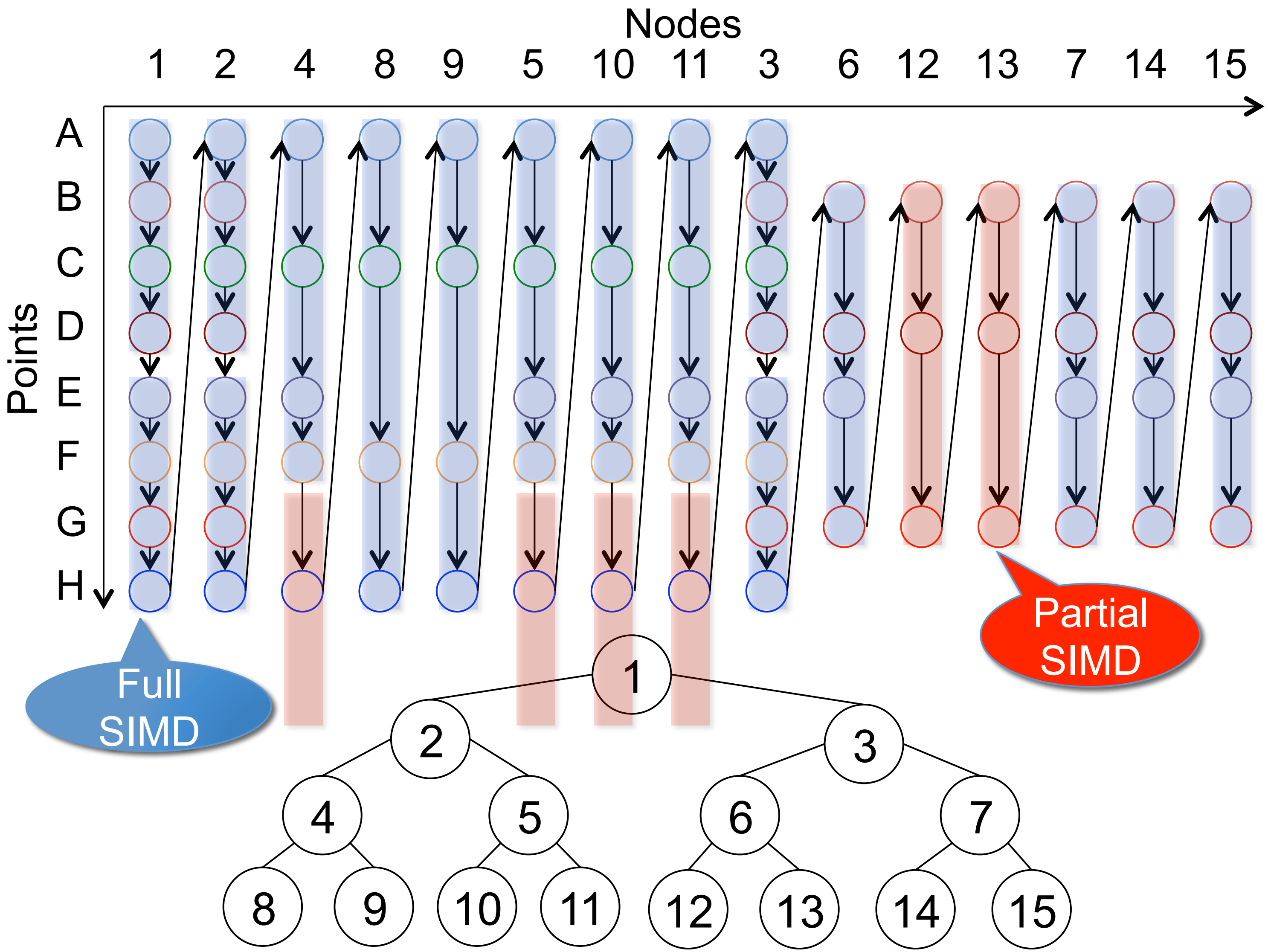
Use block size larger than SIMD width and compact points!

Use larger block size

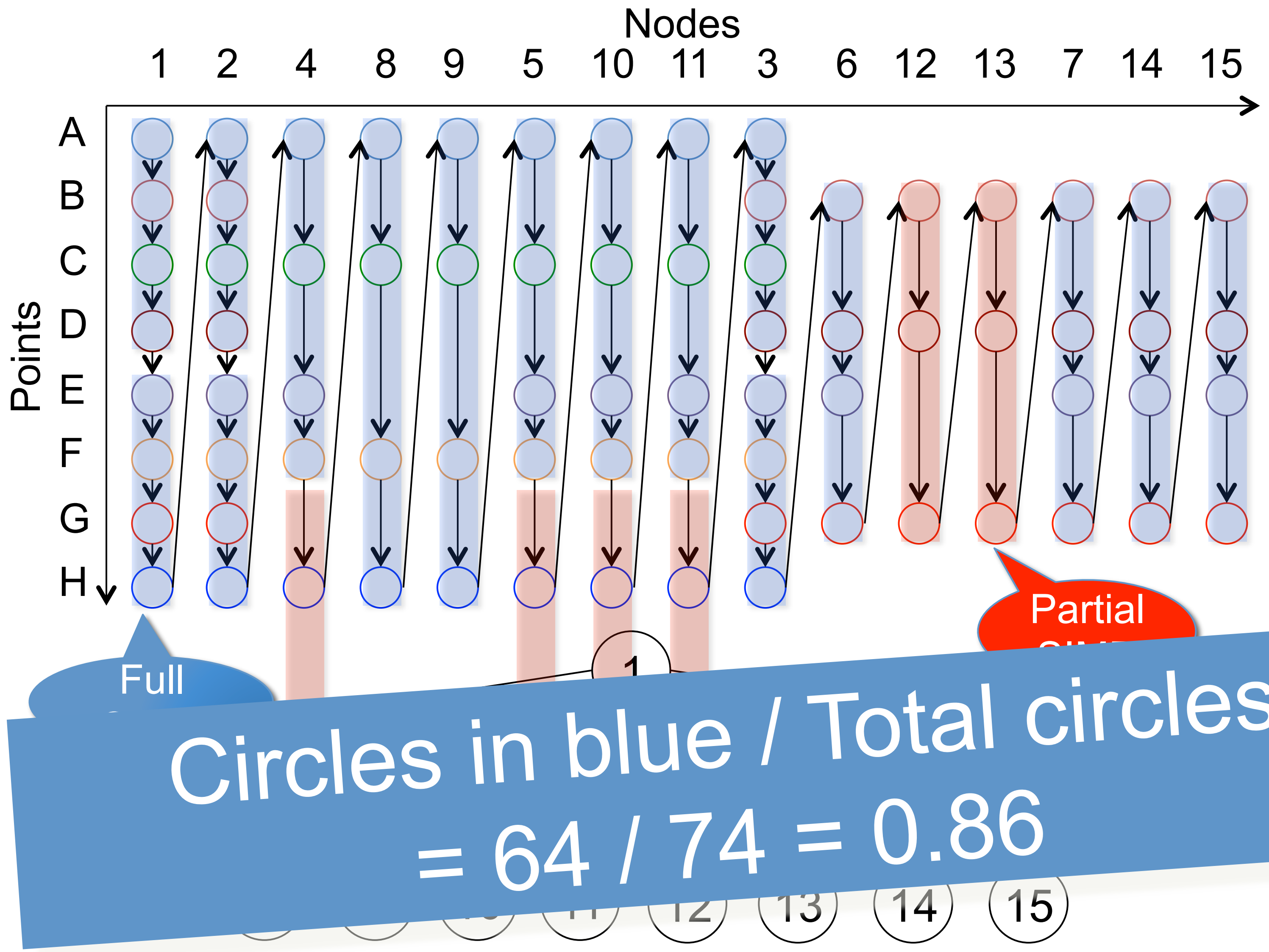




Better utilization with larger block size

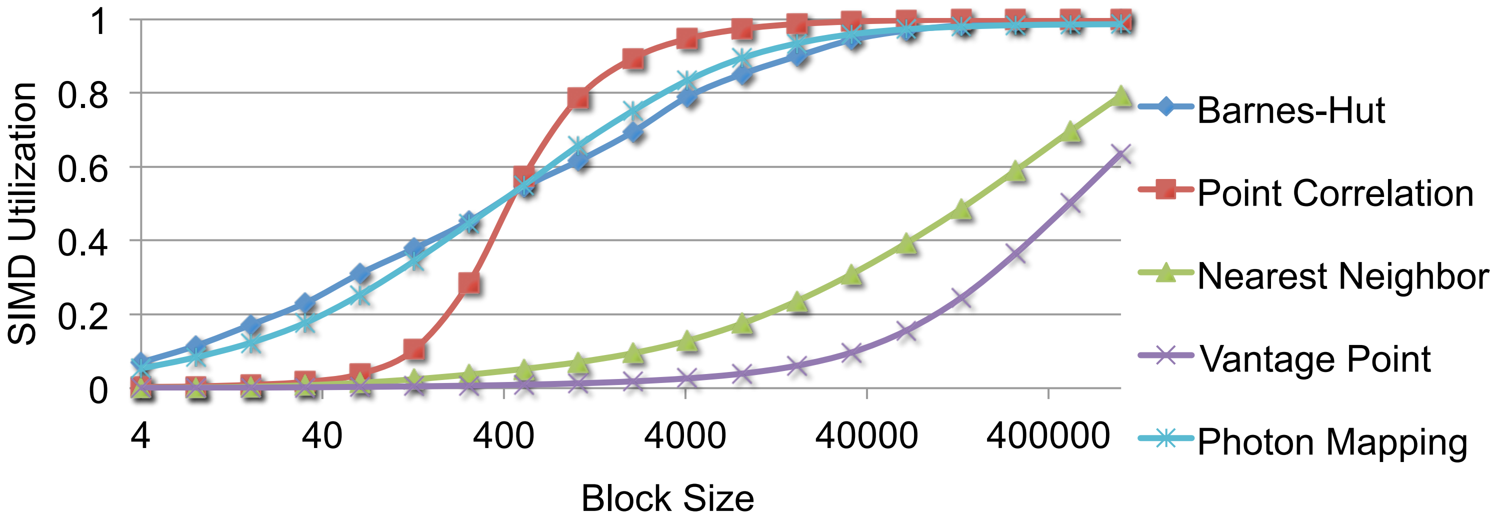


Better utilization with larger block size

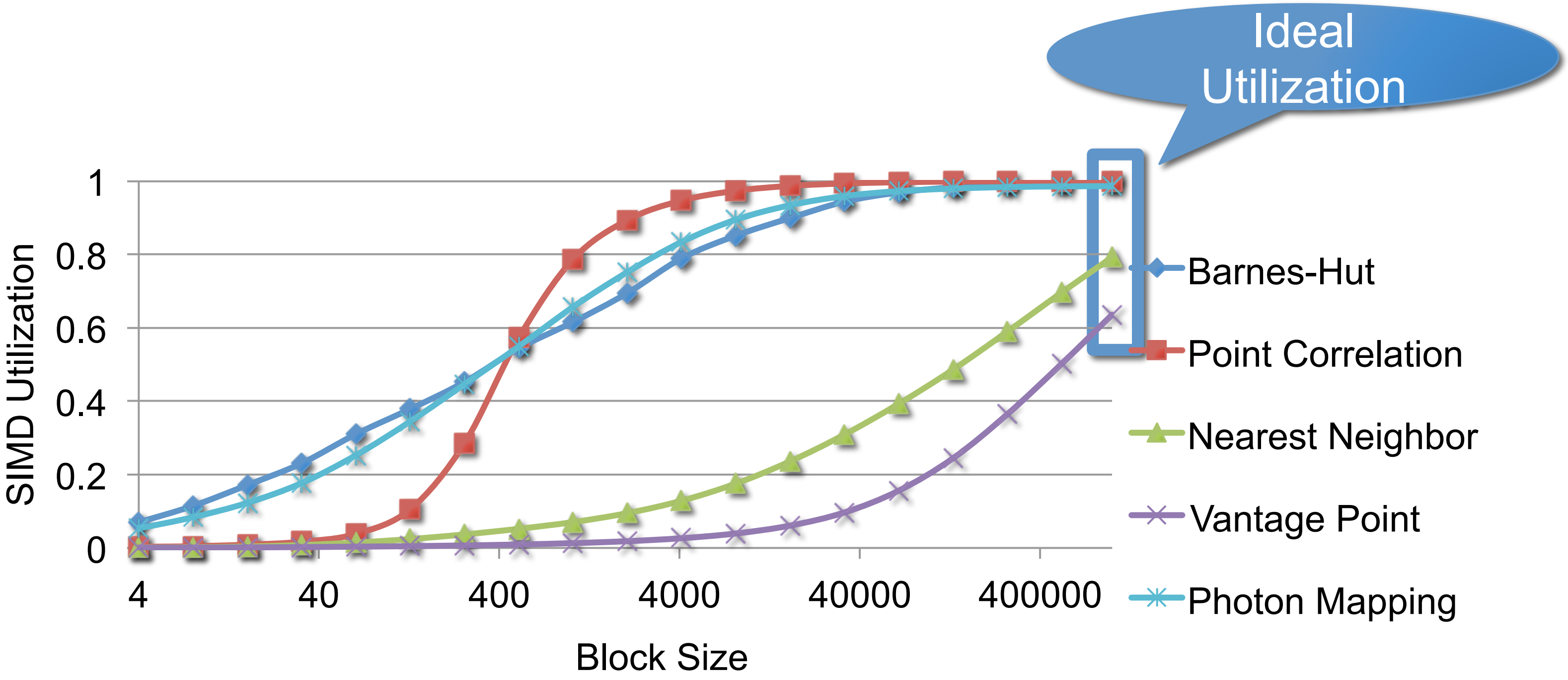


Circles in blue / Total circles
= 64 / 74 = 0.86

SIMD utilization – Block size

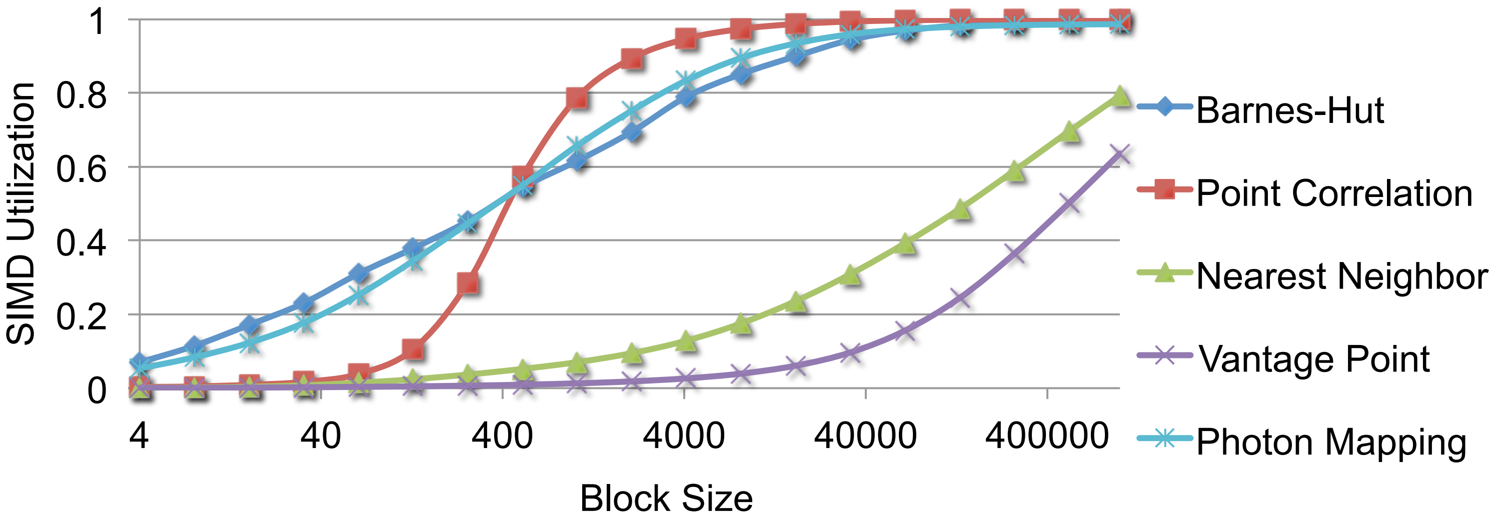


Ideal utilization

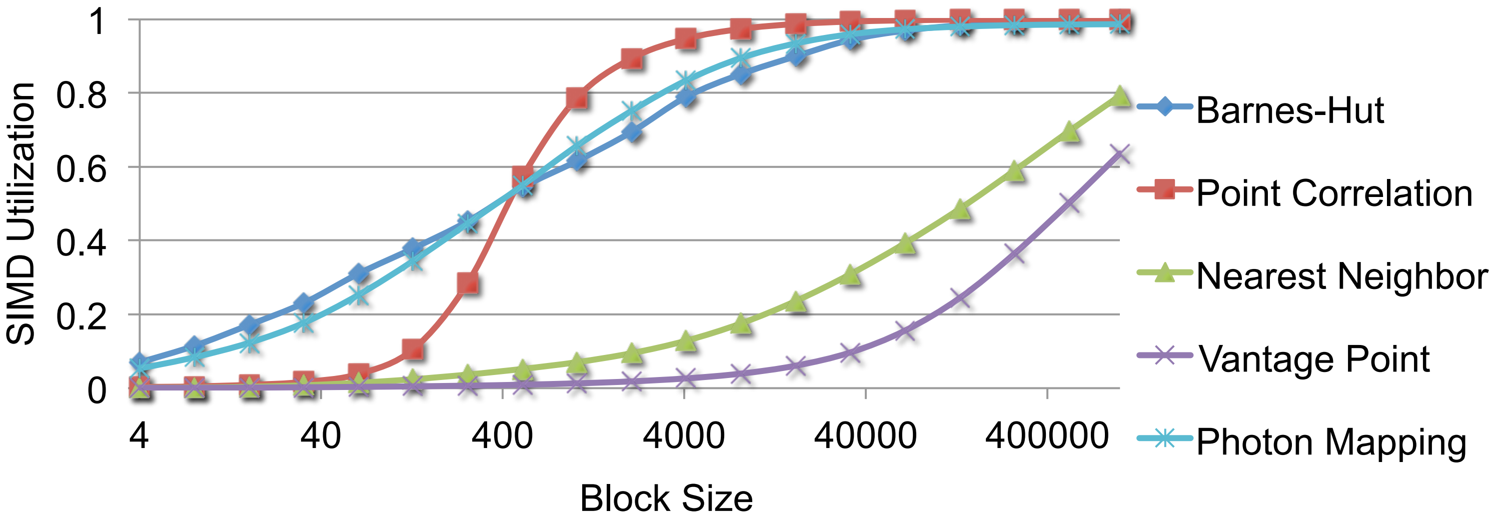


Block size equal to total points yields ideal SIMD utilization

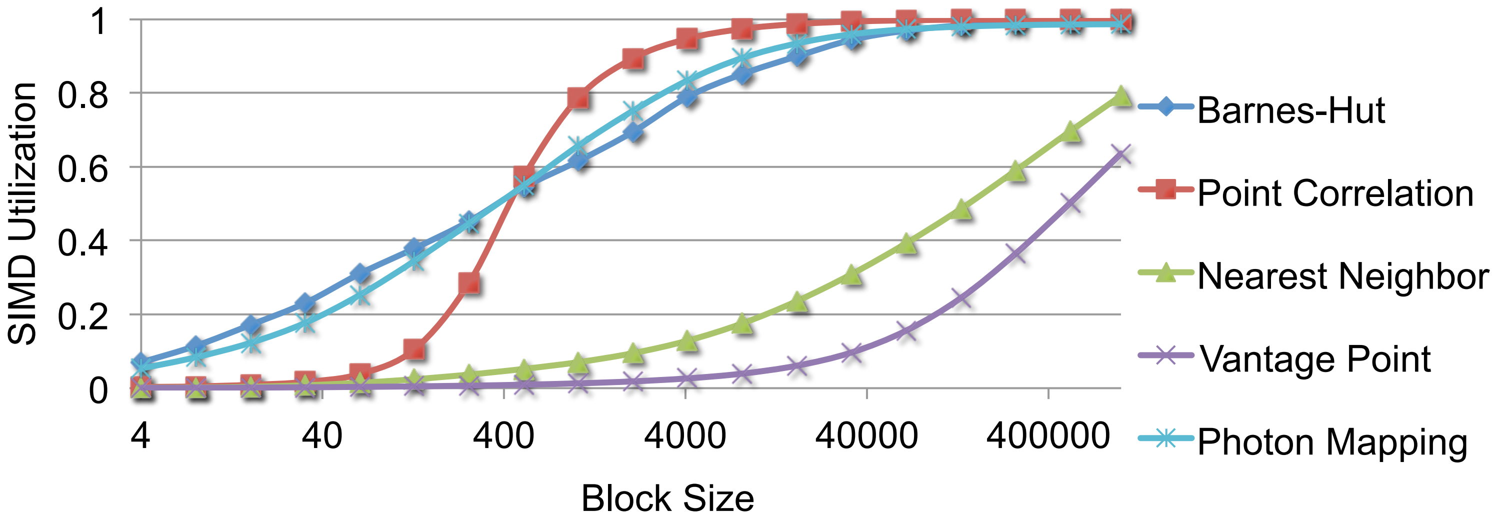
Use max block! Problem solved?



Large block has poor locality



Large block has poor locality

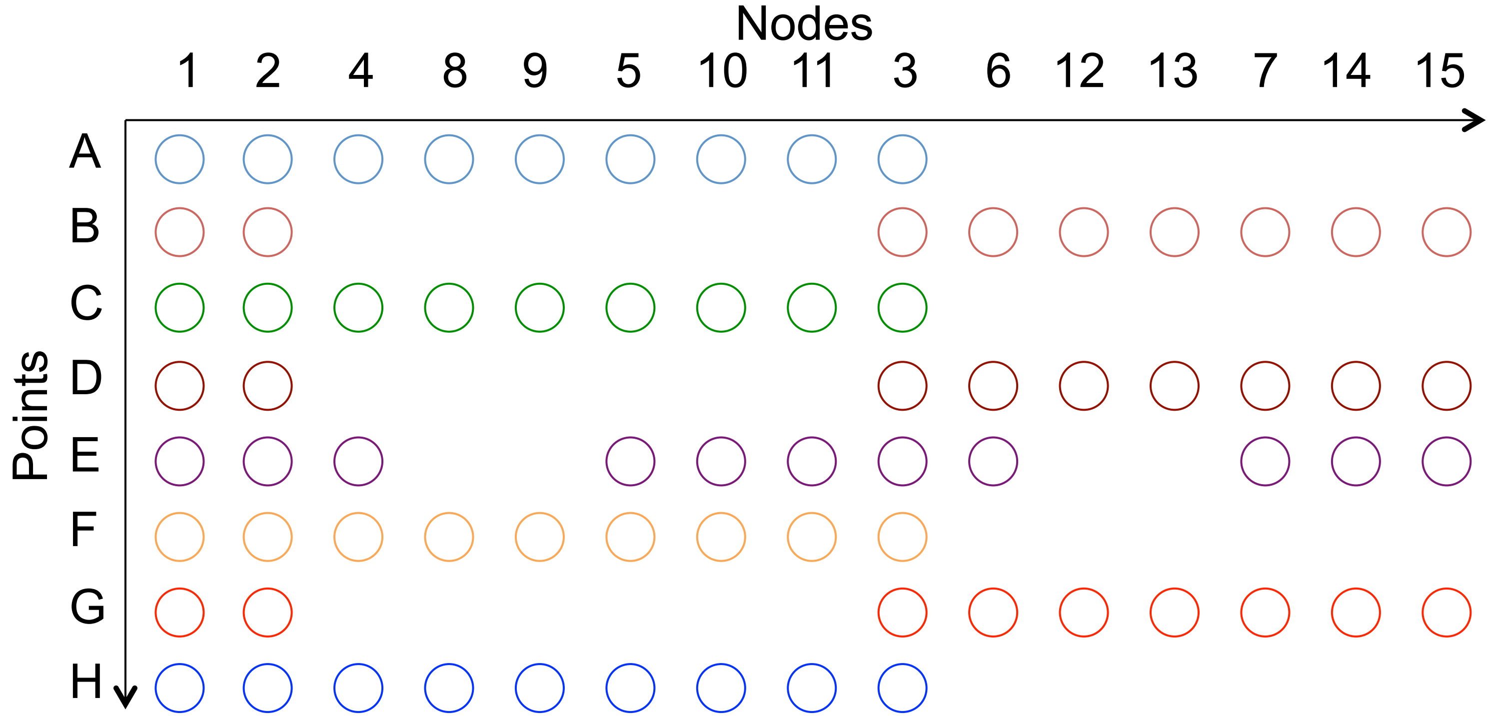


Need schedule with good utilization and good locality

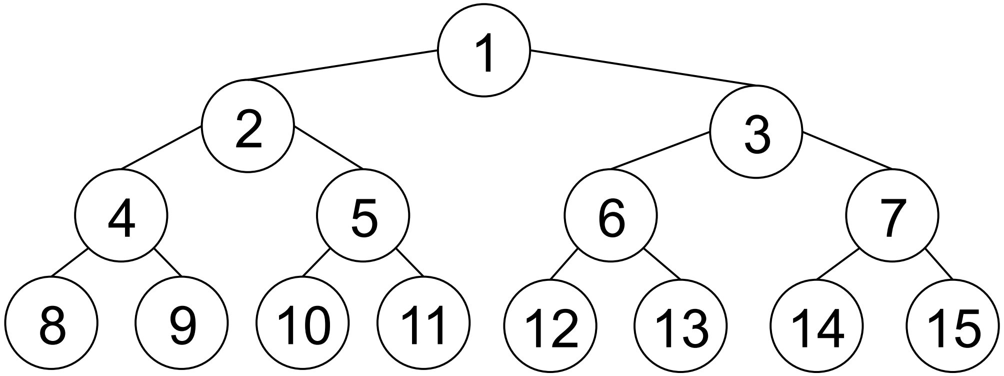
Outline

- Example & Abstract Model
- Point Blocking to Enable SIMD
- Traversal Splicing to Enhance Utilization
- Automatic Transformation
- Evaluation and Conclusion

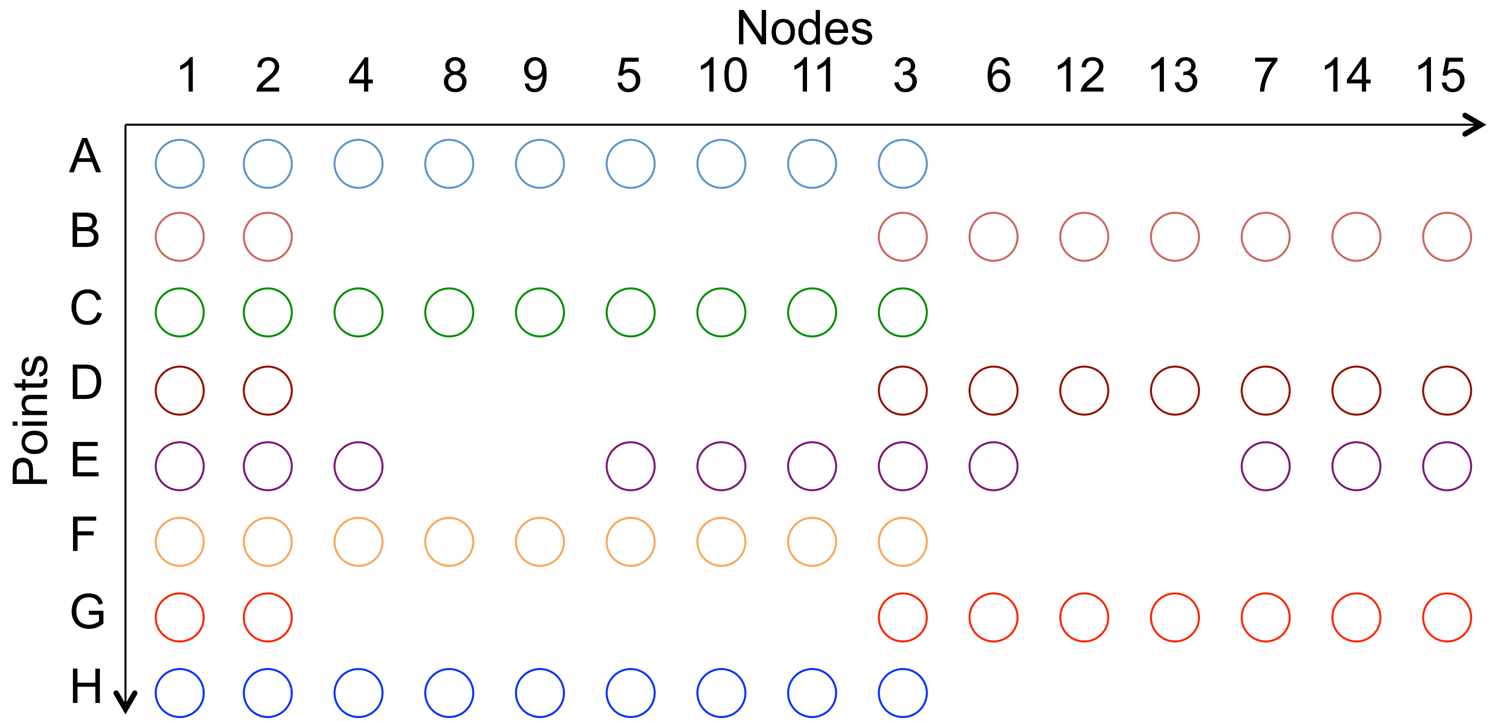
Traversal splicing [OOPSLA 2012]



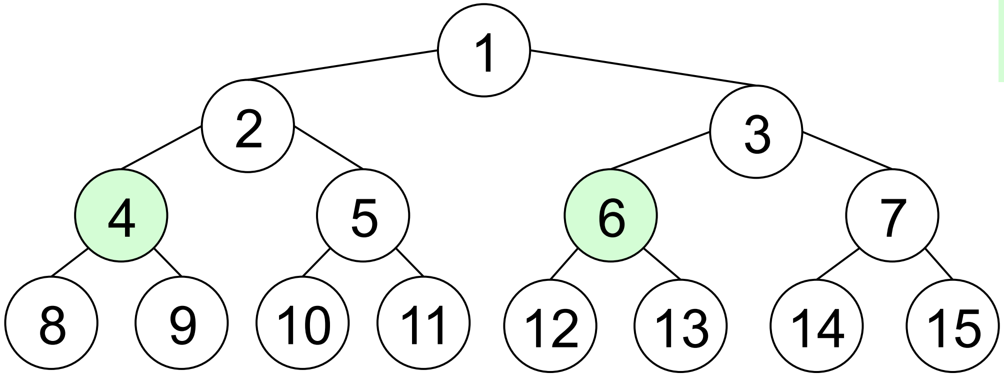
Points \ Nodes	1	2	4	8	9	5	10	11	3	6	12	13	7	14	15
A	○	○	○	○	○	○	○	○	○						
B	○	○							○	○	○	○	○	○	○
C	○	○	○	○	○	○	○	○	○						
D	○	○							○	○	○	○	○	○	○
E	○	○	○			○	○	○	○	○			○	○	○
F	○	○	○	○	○	○	○	○	○						
G	○	○							○	○	○	○	○	○	○
H	○	○	○	○	○	○	○	○	○						



Traversal splicing [OOPSLA 2012]

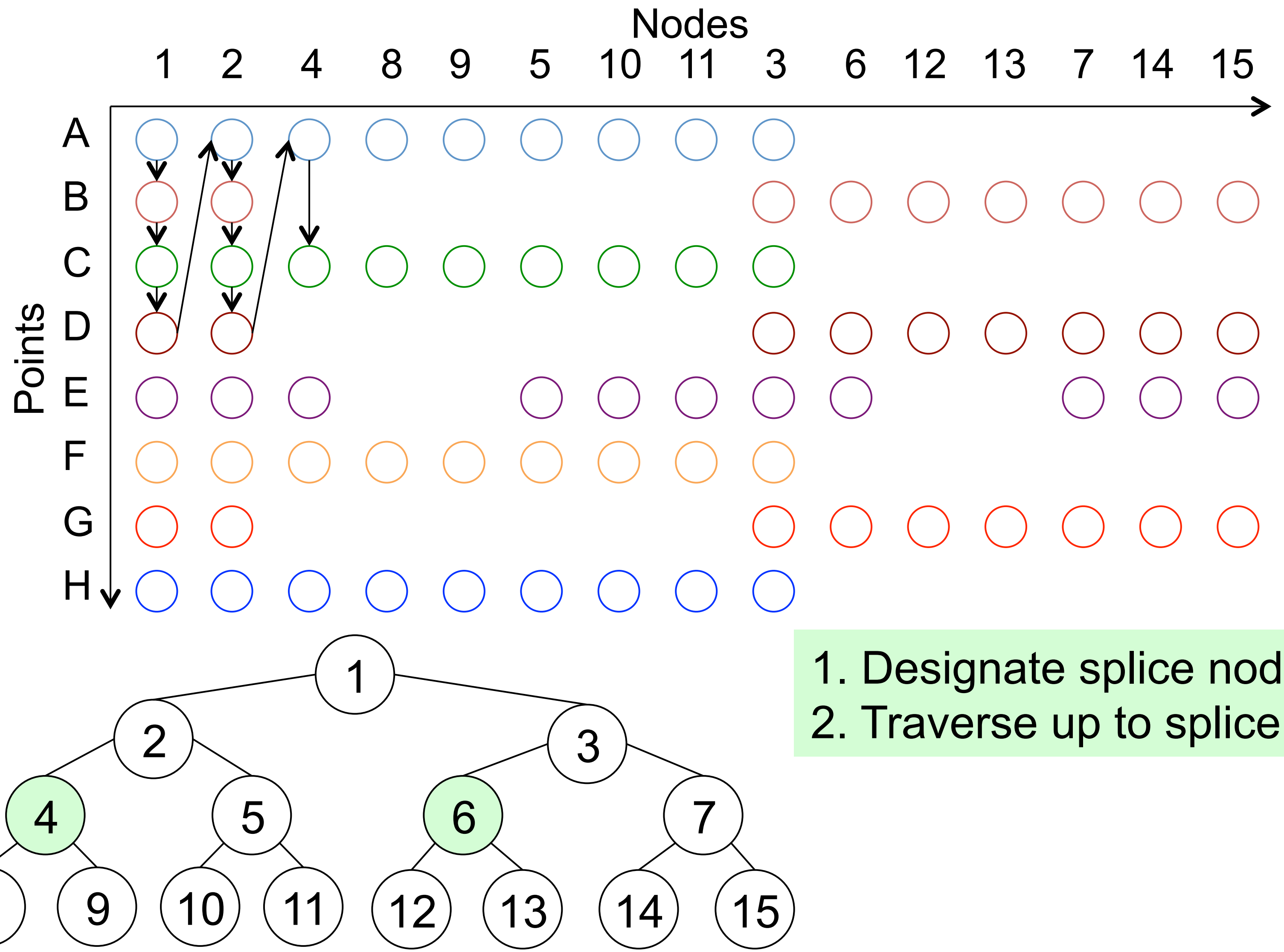


	1	2	4	8	9	5	10	11	3	6	12	13	7	14	15
A	○	○	○	○	○	○	○	○	○						
B	○	○							○	○	○	○	○	○	○
C	○	○	○	○	○	○	○	○	○						
D	○	○							○	○	○	○	○	○	○
E	○	○	○			○	○	○	○	○			○	○	○
F	○	○	○	○	○	○	○	○	○						
G	○	○							○	○	○	○	○	○	○
H	○	○	○	○	○	○	○	○	○						



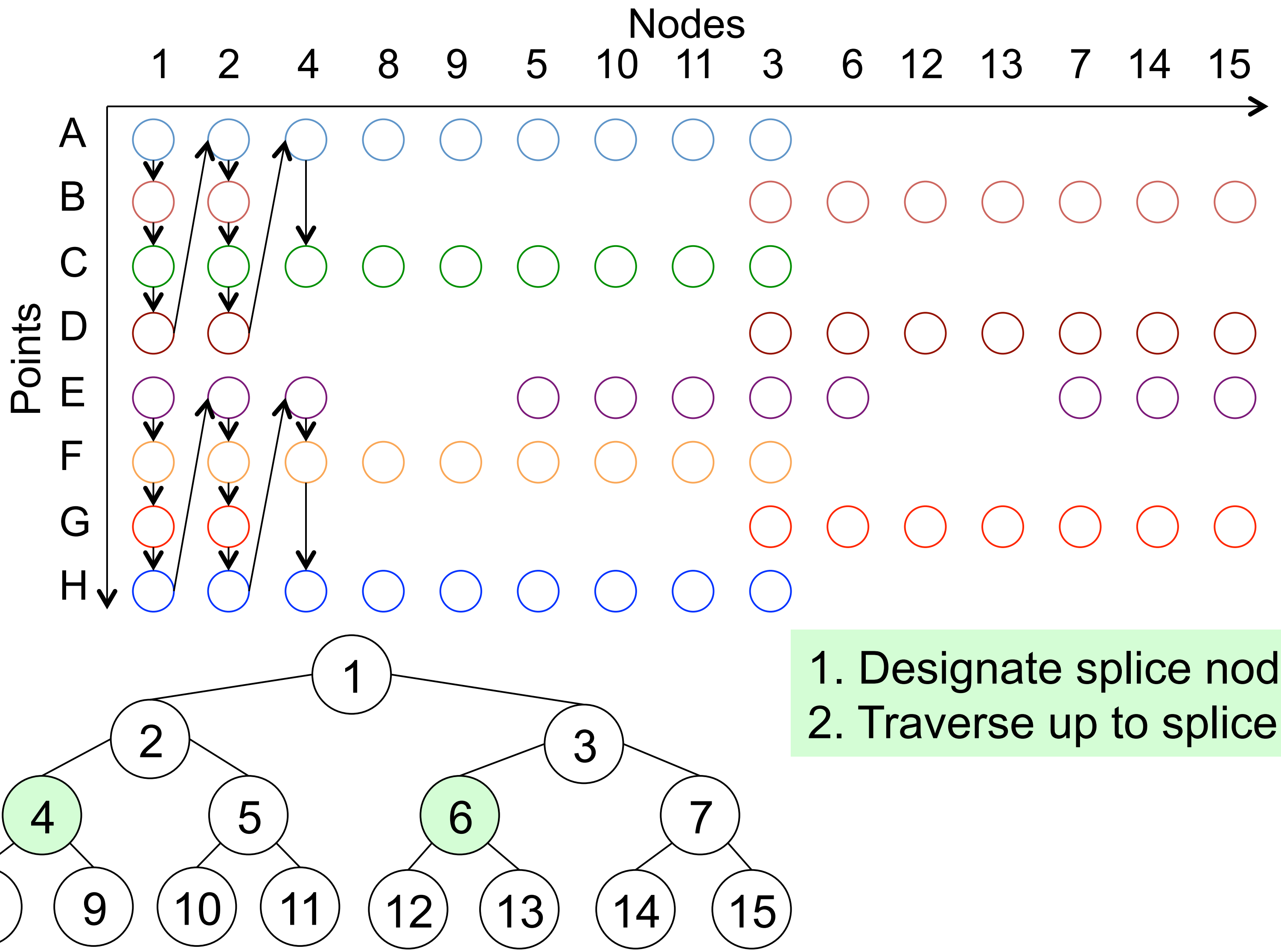
1. Designate splice nodes

Traversal splicing [OOPSLA 2012]



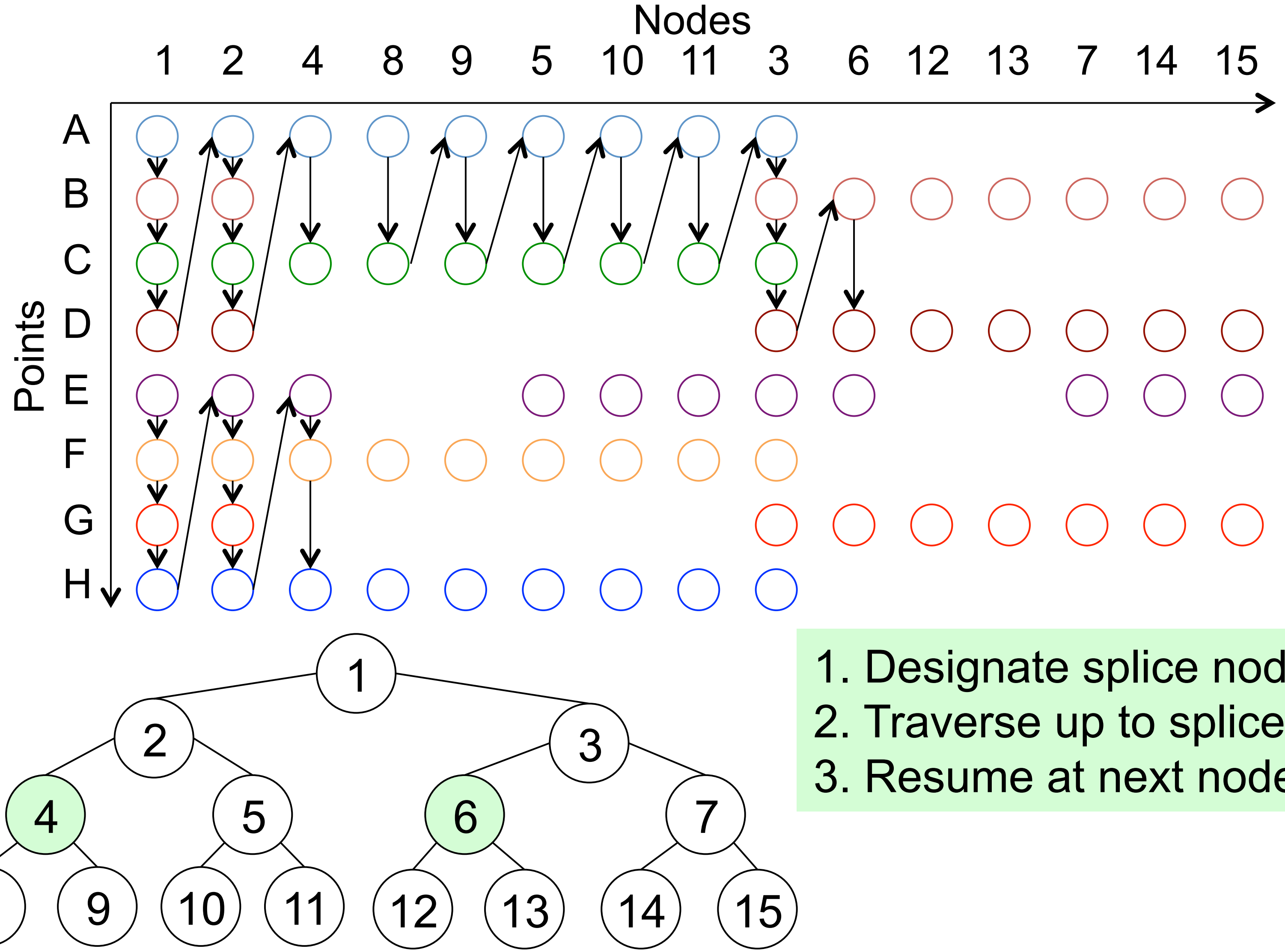
1. Designate splice nodes
2. Traverse up to splice node

Traversal splicing [OOPSLA 2012]



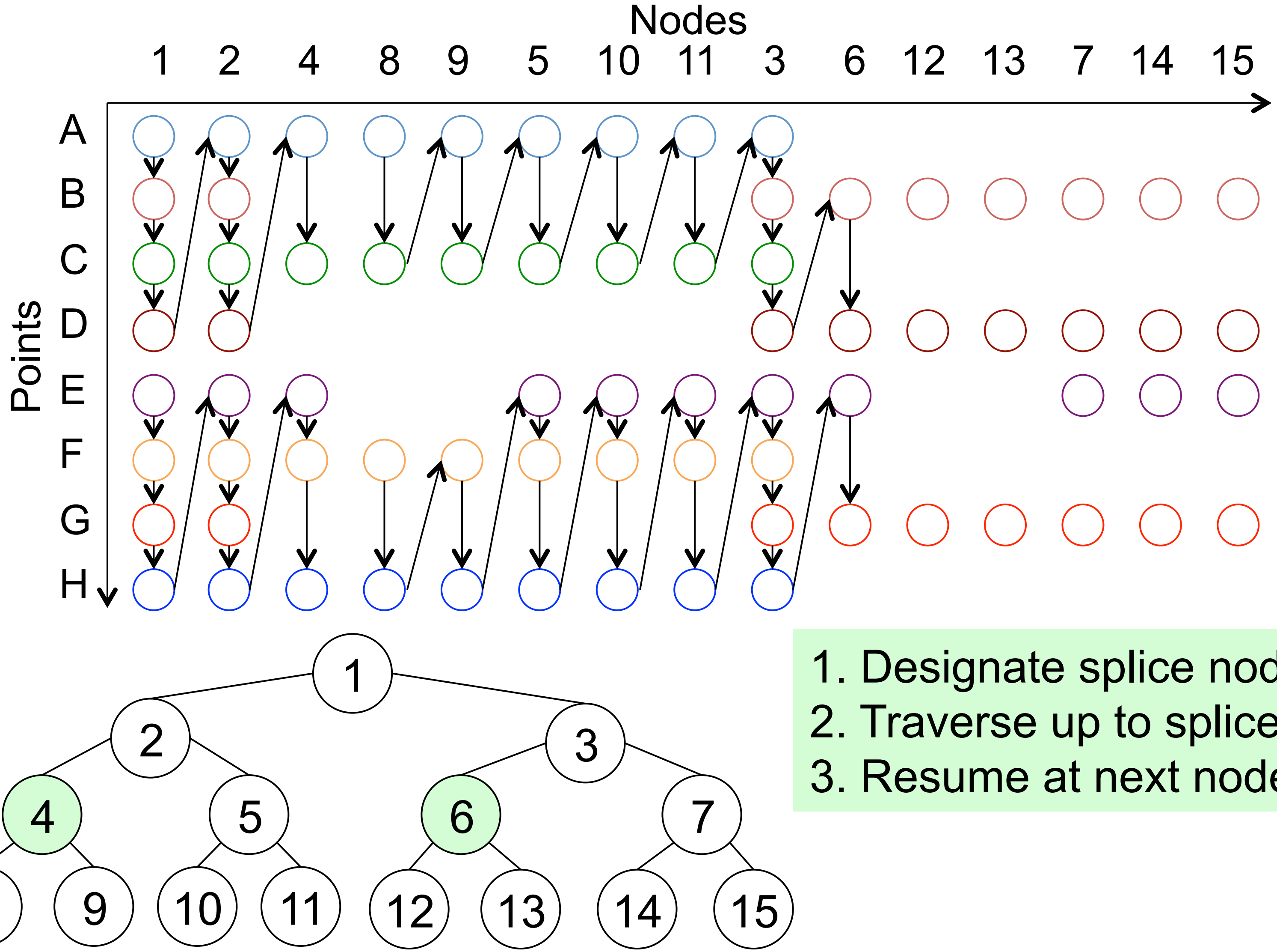
1. Designate splice nodes
2. Traverse up to splice node

Traversal splicing [OOPSLA 2012]



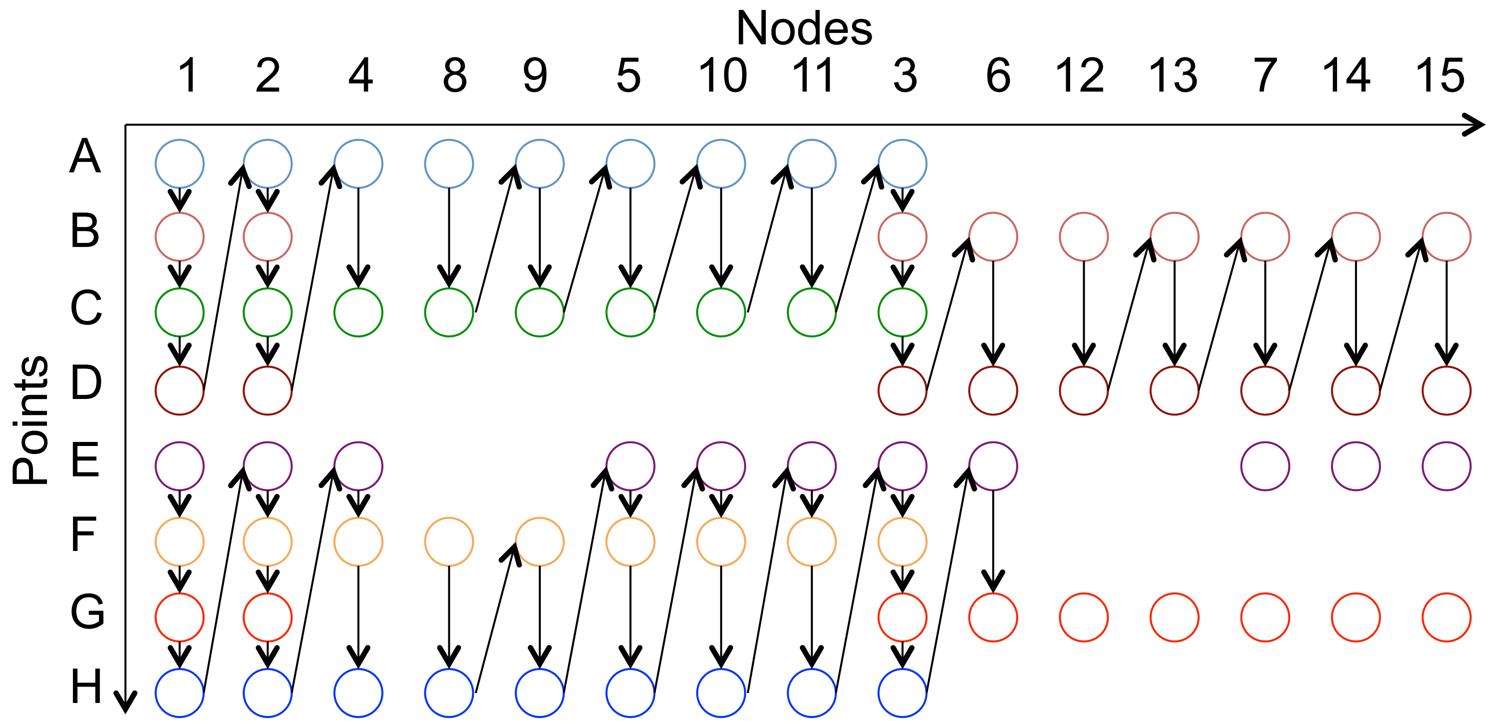
1. Designate splice nodes
2. Traverse up to splice node
3. Resume at next node

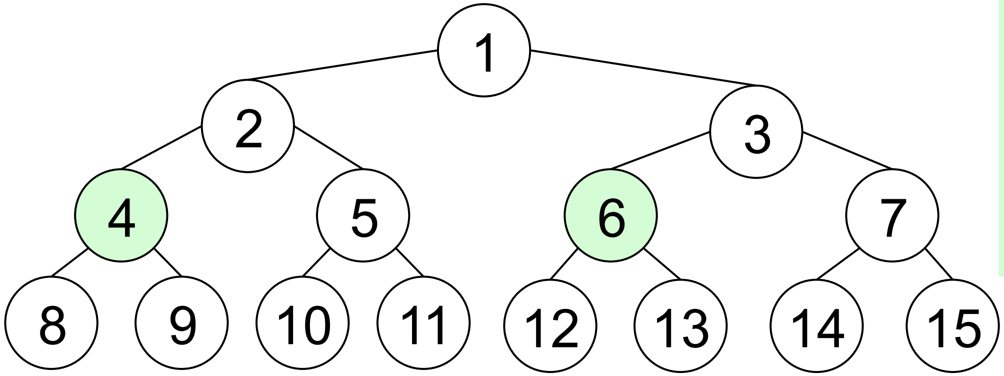
Traversal splicing [OOPSLA 2012]



1. Designate splice nodes
2. Traverse up to splice node
3. Resume at next node

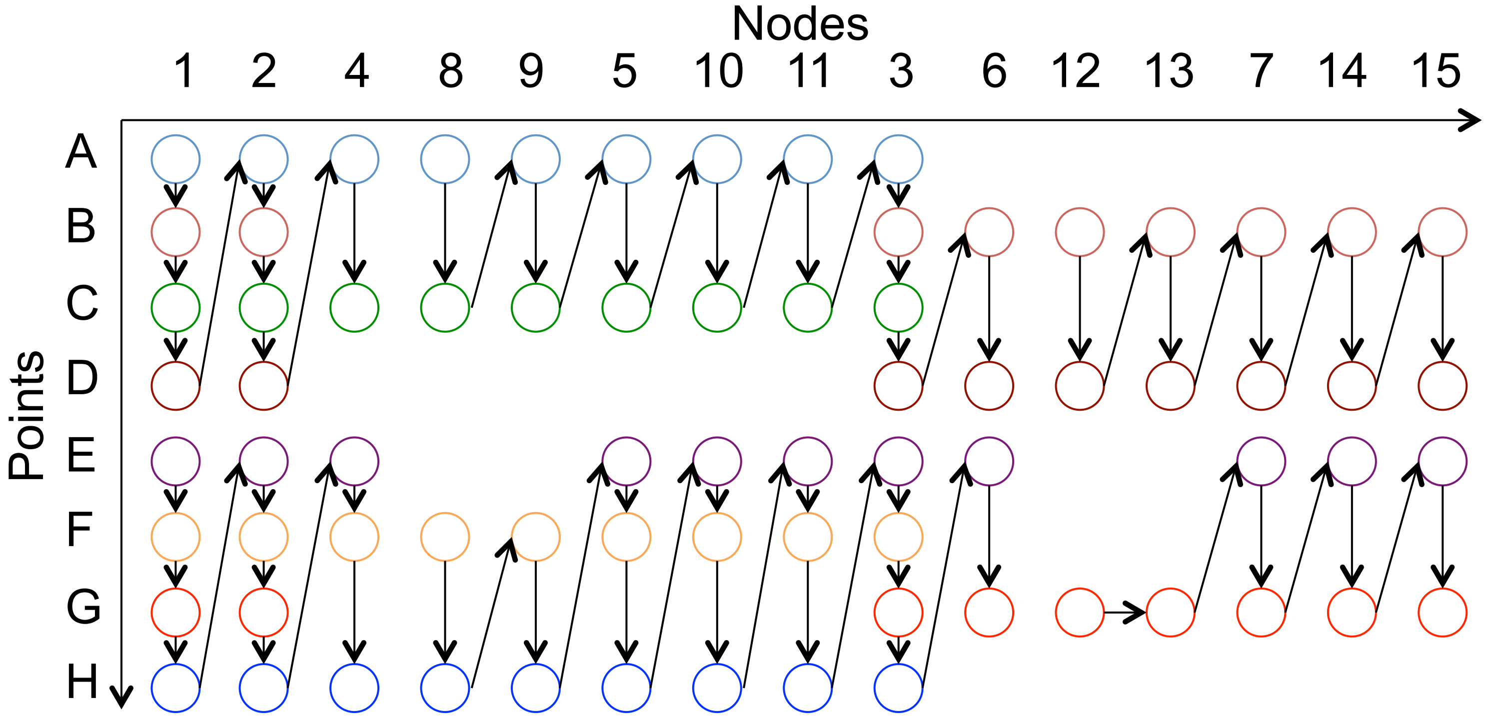
Traversal splicing [OOPSLA 2012]

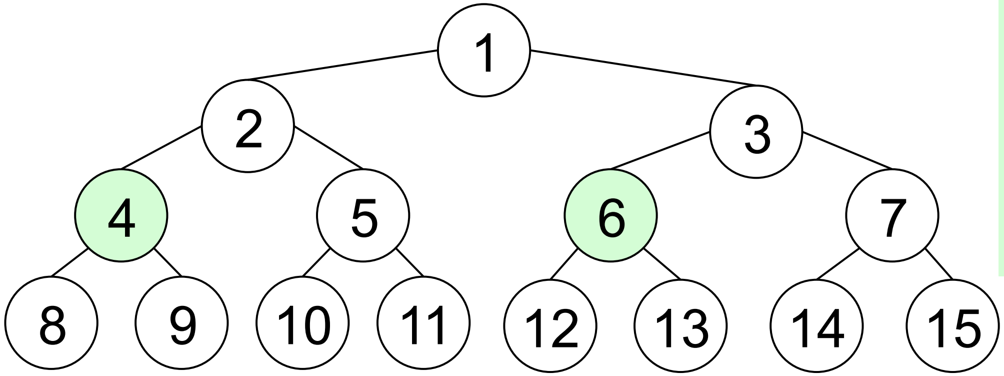




1. Designate splice nodes
2. Traverse up to splice node
3. Resume at next node
4. Repeat 2-3 until finished

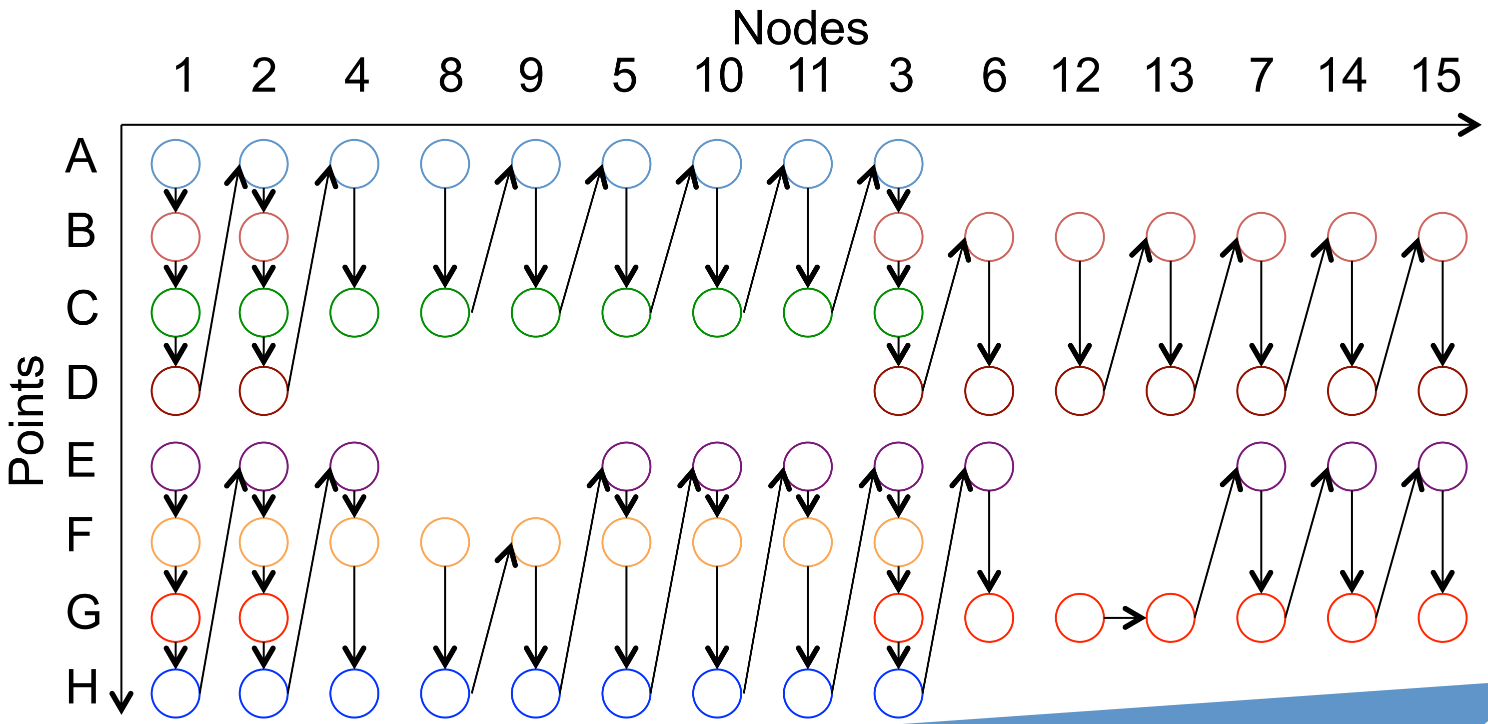
Traversal splicing [OOPSLA 2012]





1. Designate splice nodes
2. Traverse up to splice node
3. Resume at next node
4. Repeat 2-3 until finished

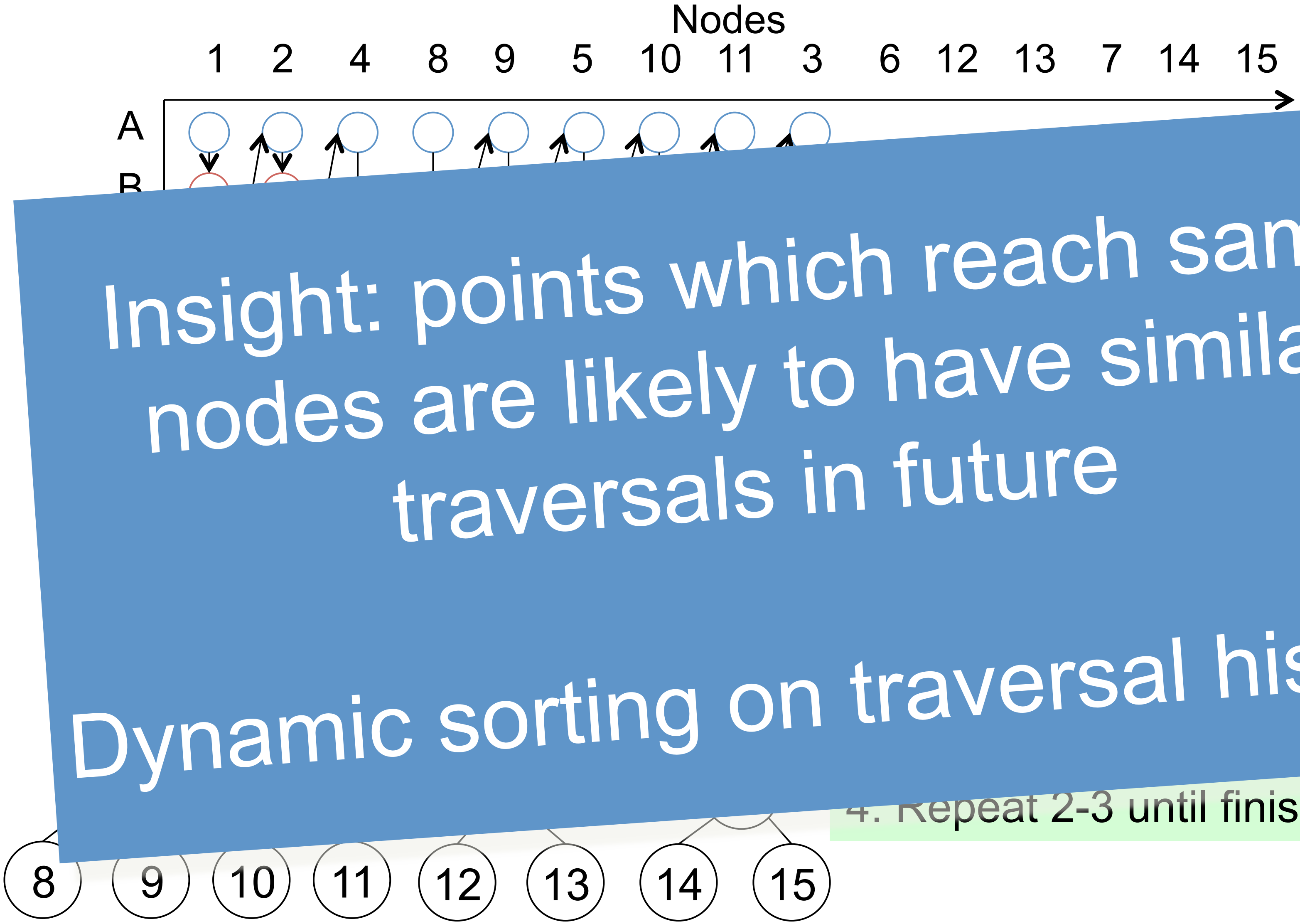
Can change order of points



We can change the order of paused points, but how?

4. Repeat 2-3 until finished

Dynamic sorting

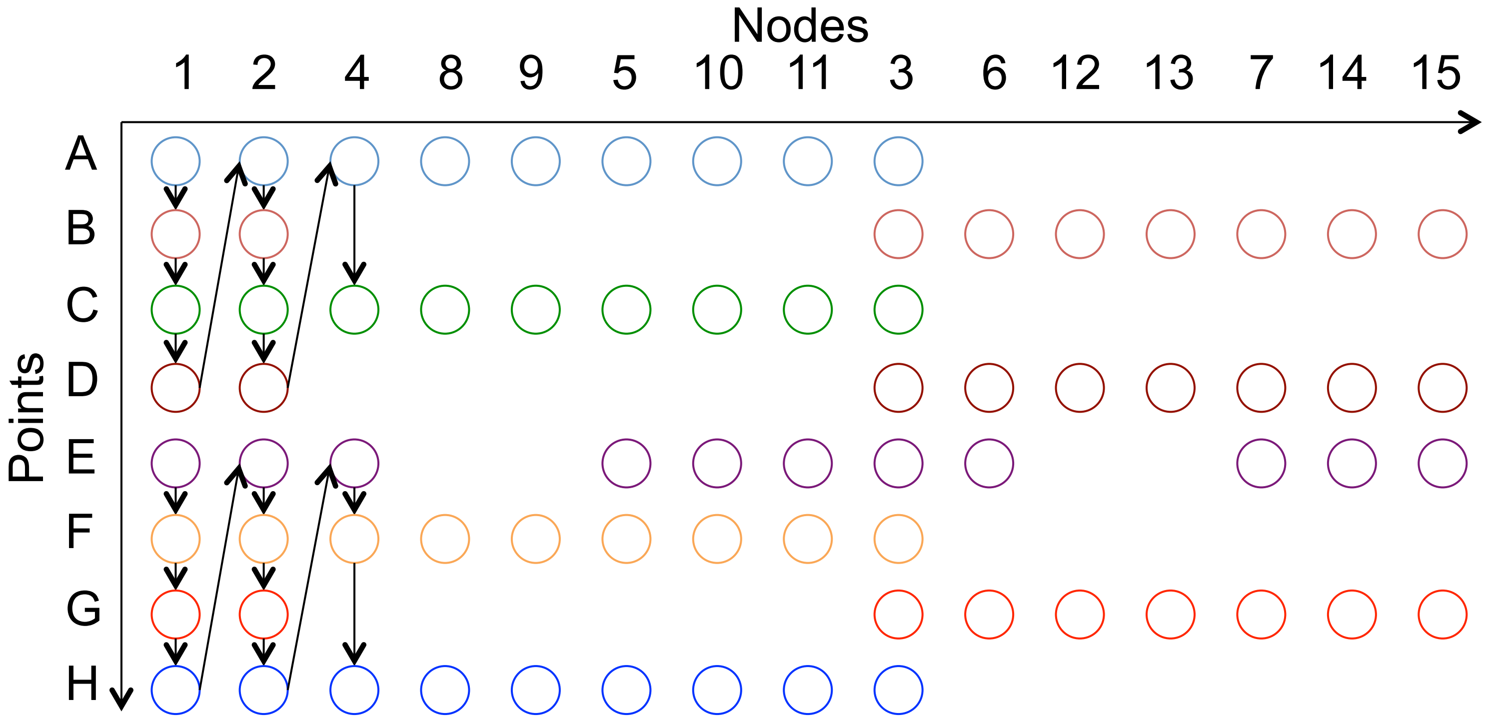


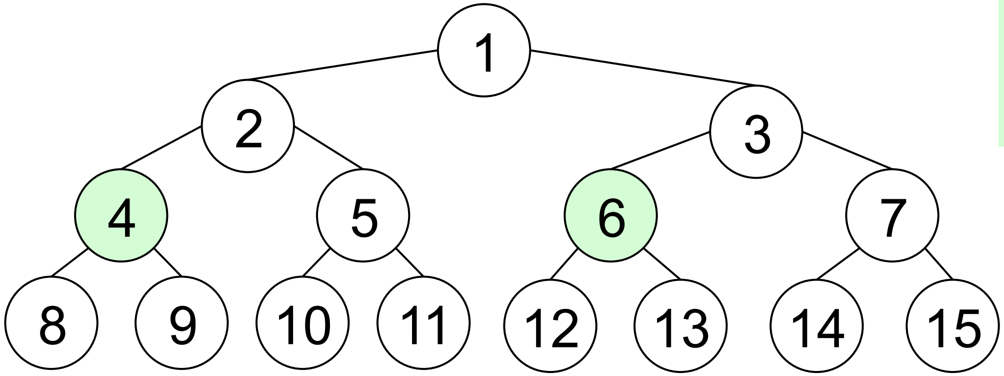
Insight: points which reach same nodes are likely to have similar traversals in future

Dynamic sorting on traversal history

4. Repeat 2-3 until finished

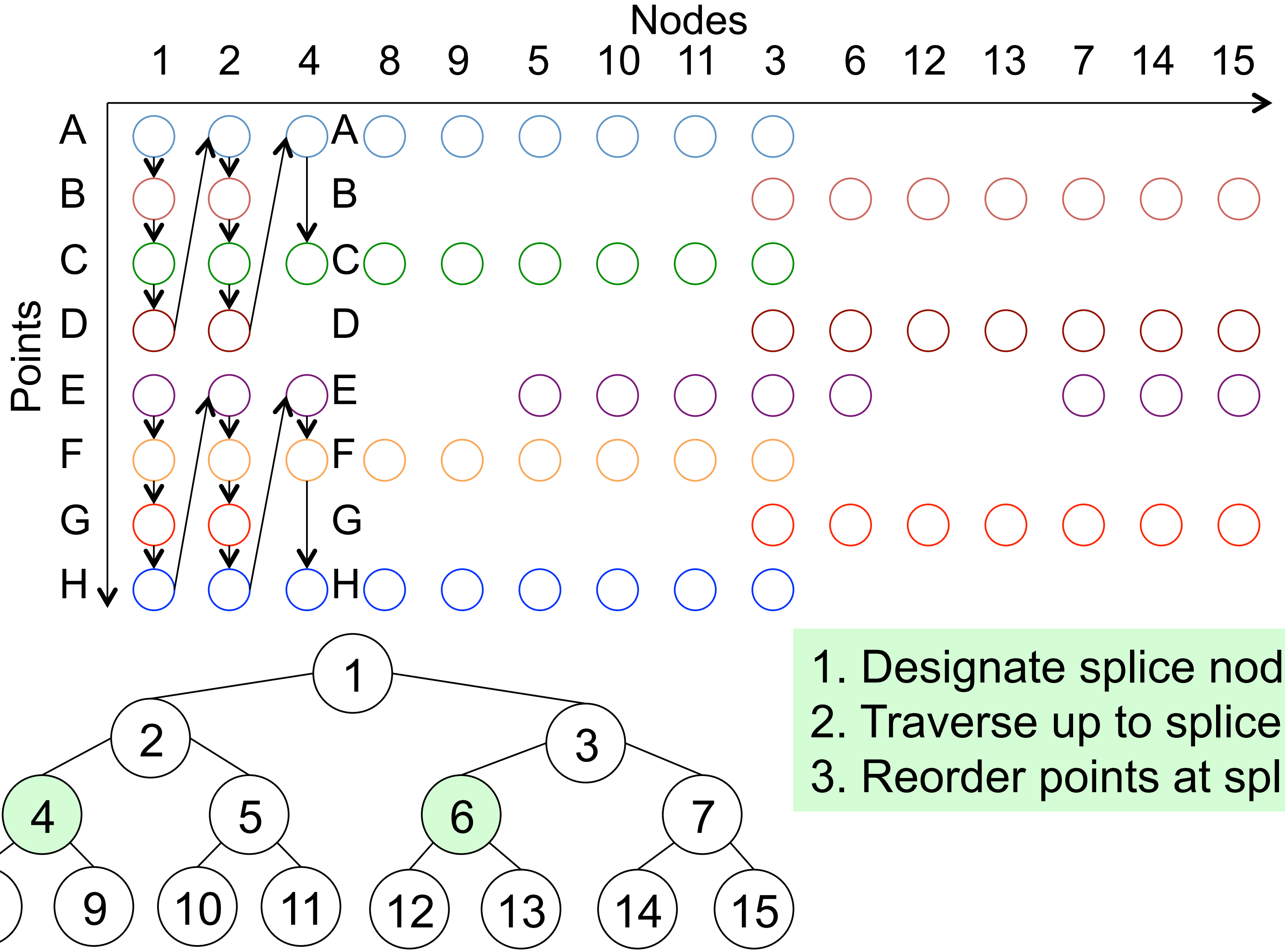
Dynamic sorting





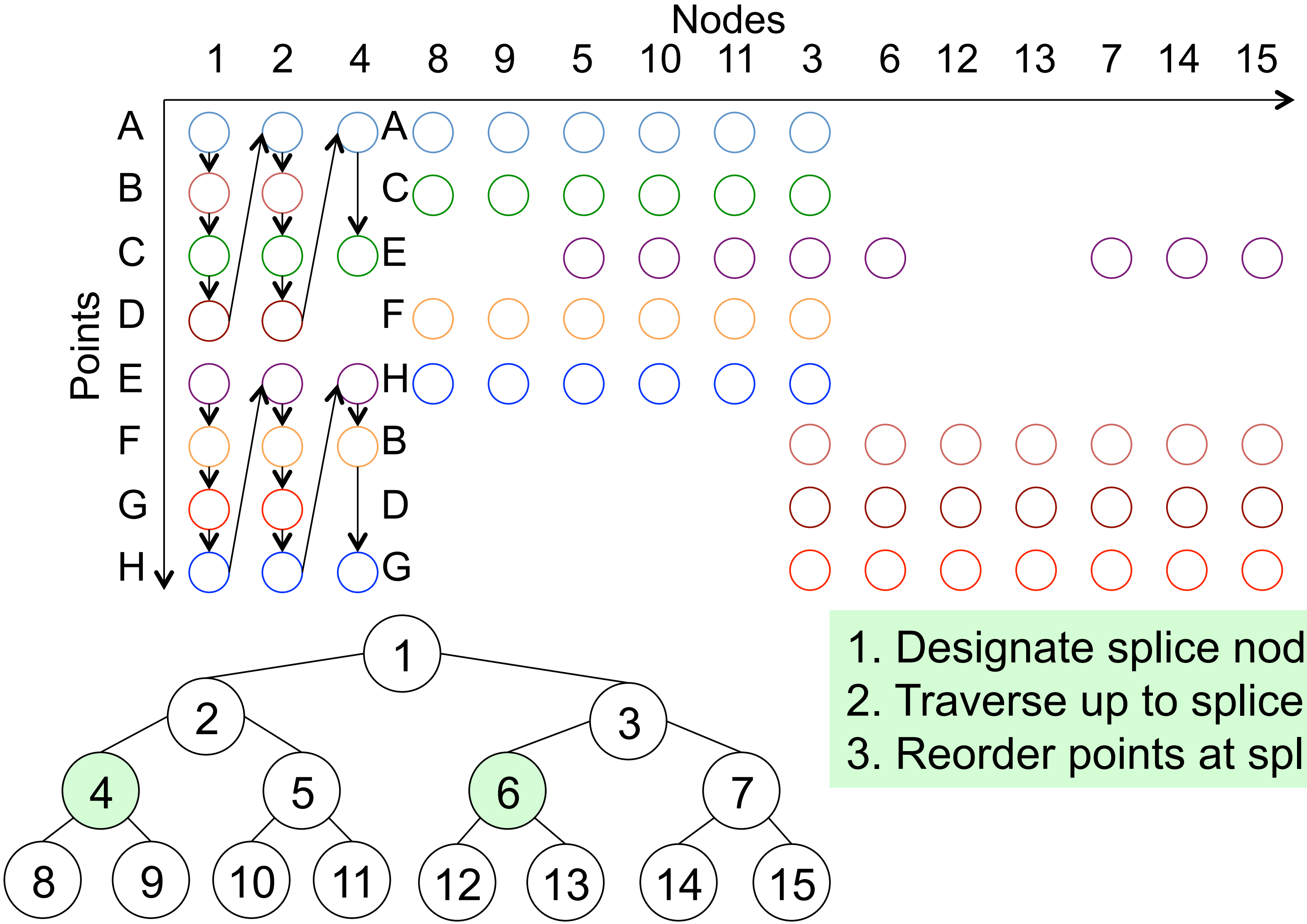
1. Designate splice nodes
2. Traverse up to splice node

Dynamic sorting



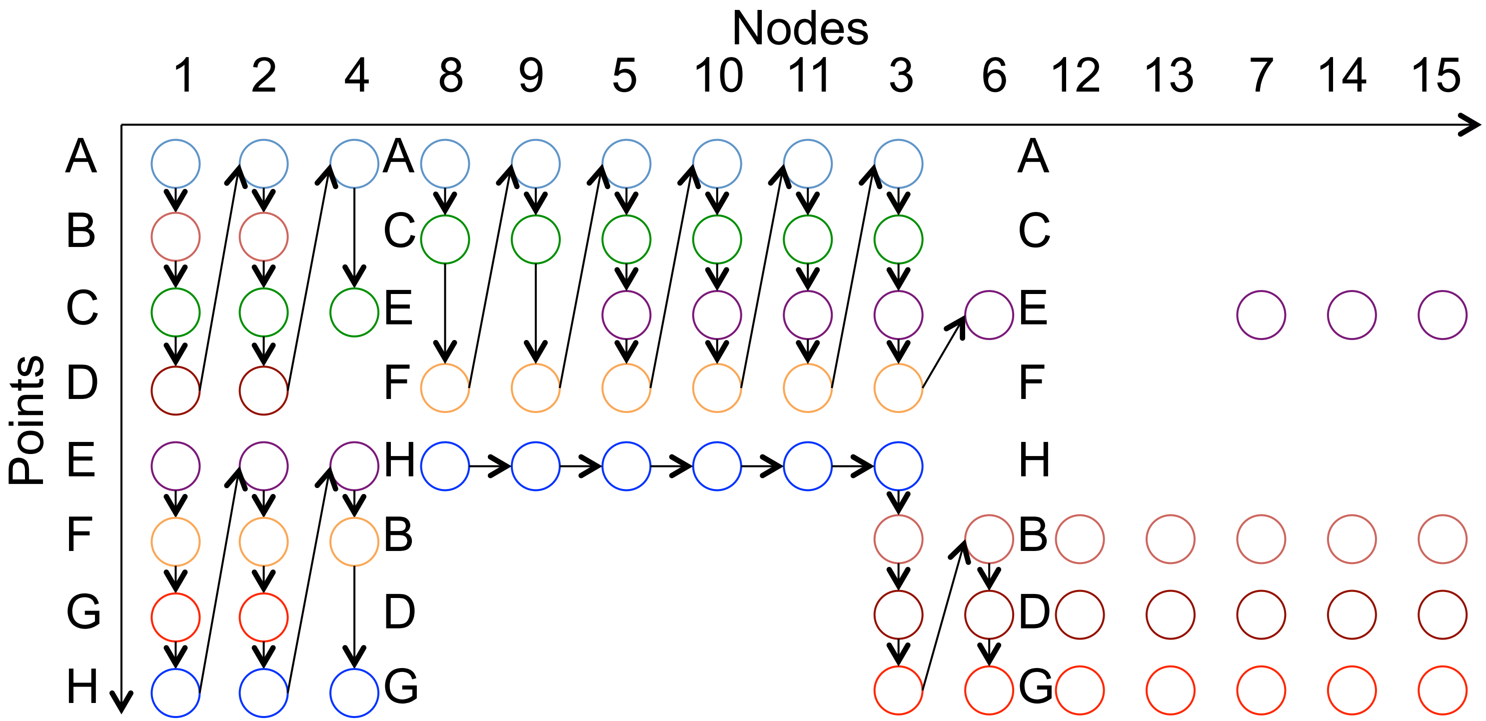
1. Designate splice nodes
2. Traverse up to splice node
3. Reorder points at splice node

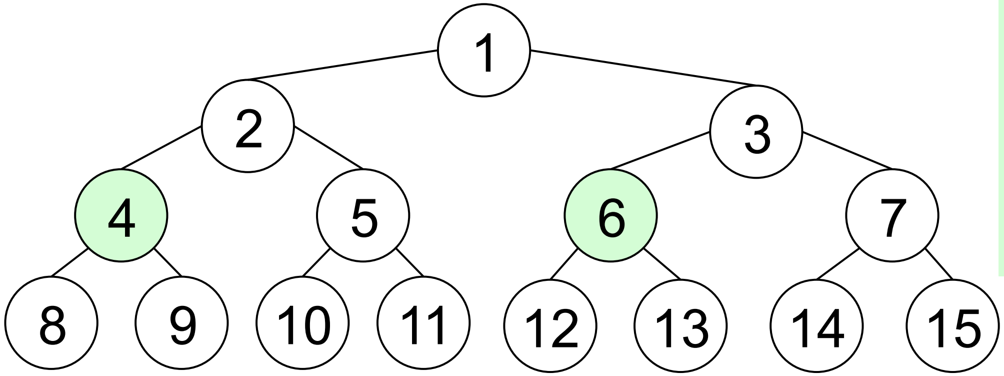
Dynamic sorting



1. Designate splice nodes
2. Traverse up to splice node
3. Reorder points at splice node

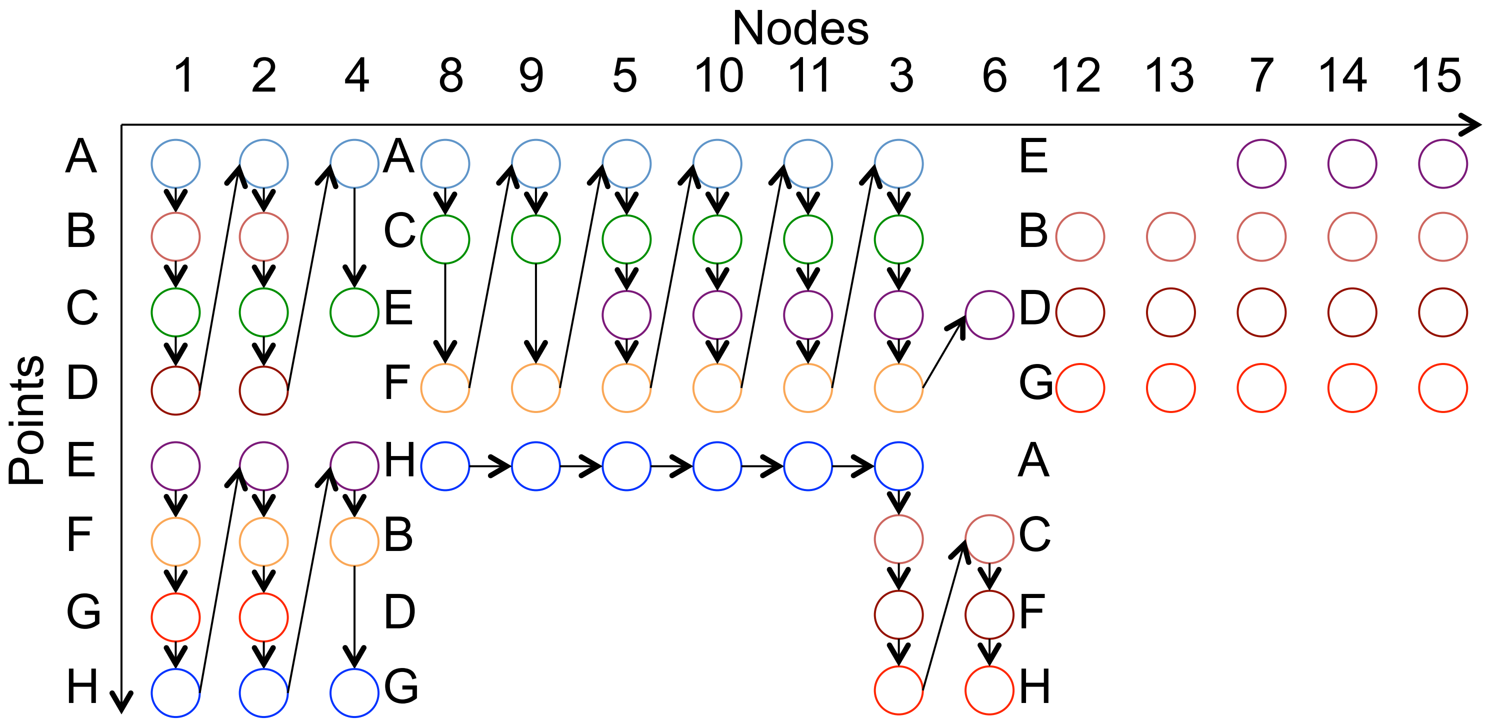
Dynamic sorting

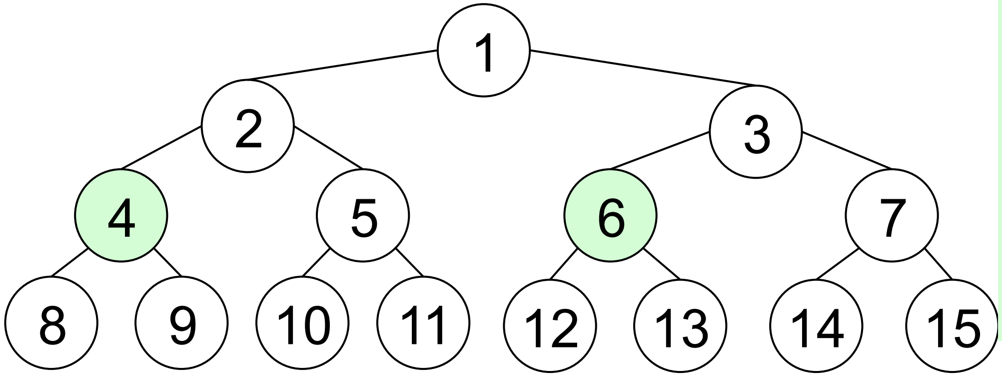




1. Designate splice nodes
2. Traverse up to splice node
3. Reorder points at splice node
4. Resume at next node

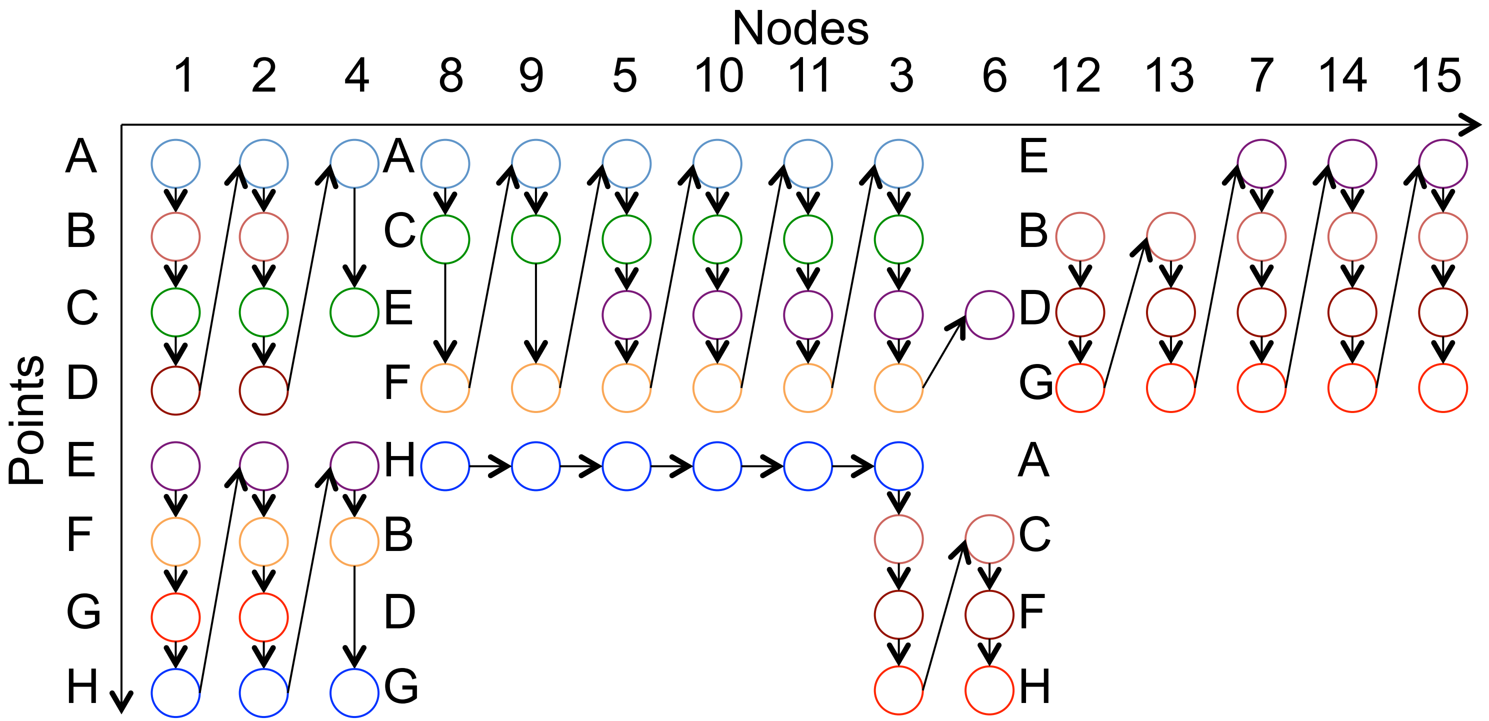
Dynamic sorting

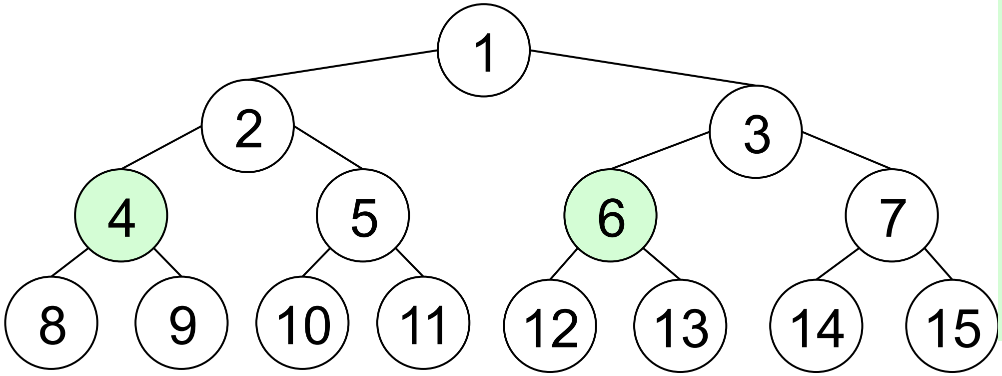




1. Designate splice nodes
2. Traverse up to splice node
3. Reorder points at splice node
4. Resume at next node
5. Repeat 2-4 until finished

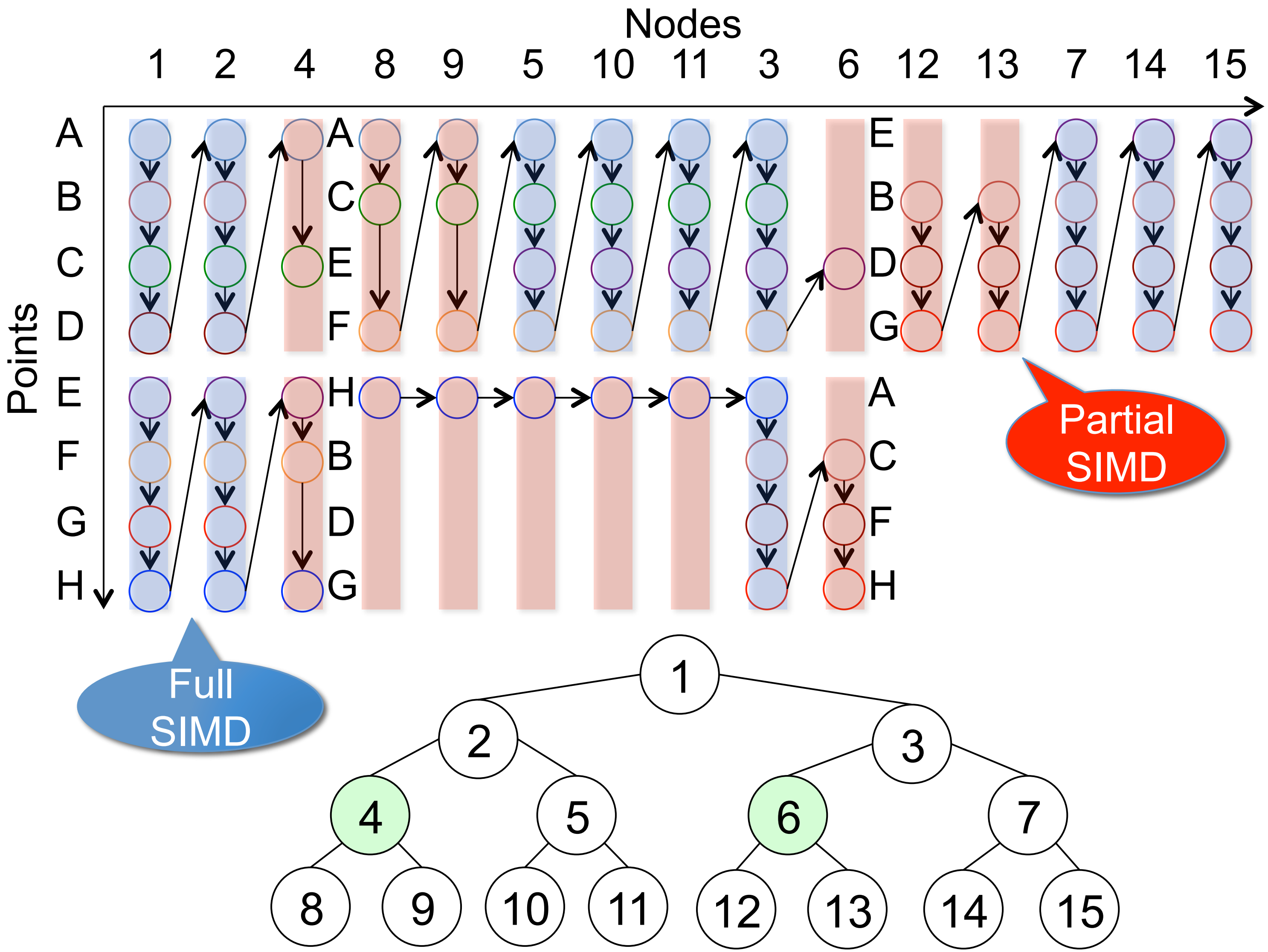
Dynamic sorting



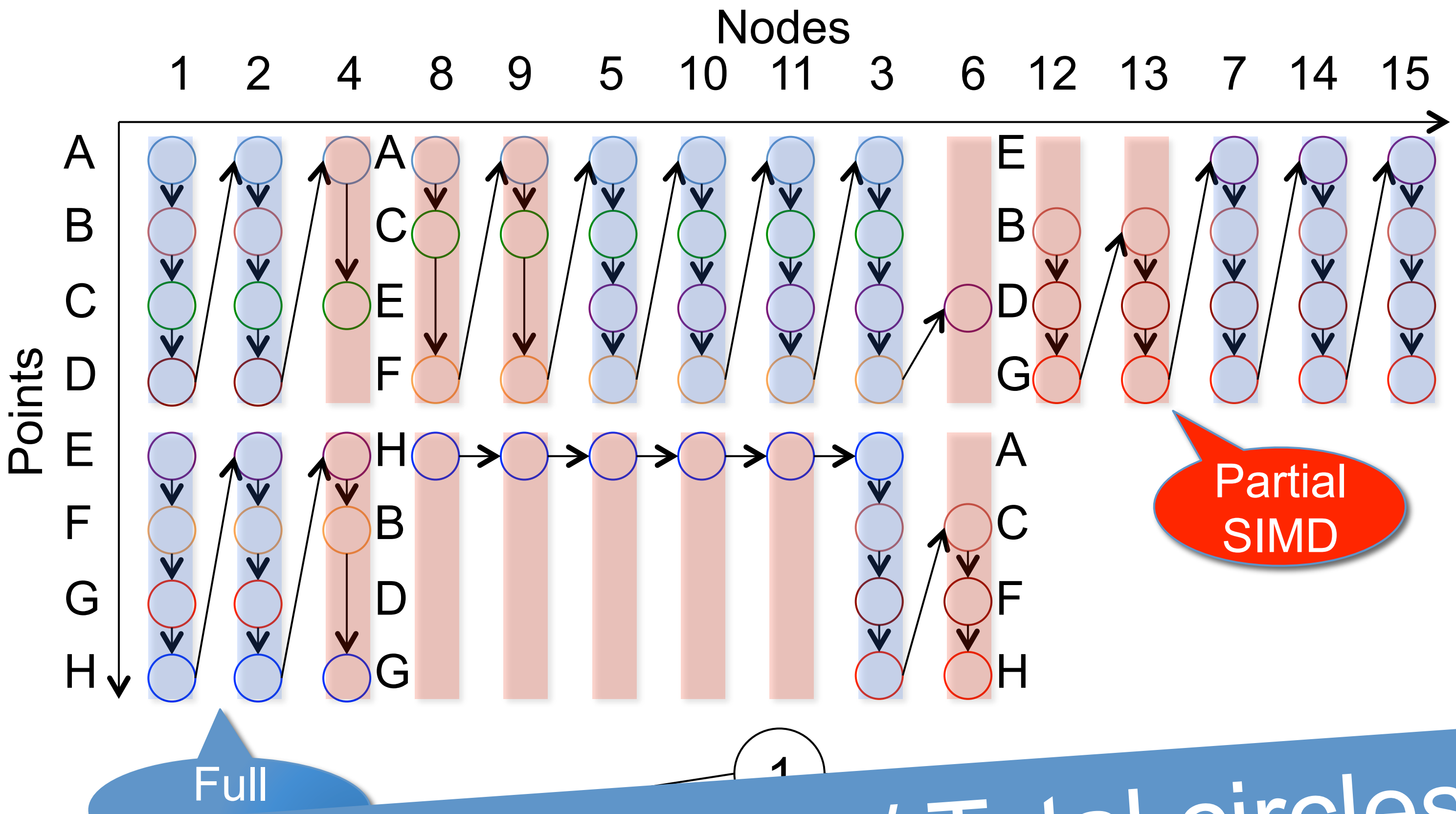


1. Designate splice nodes
2. Traverse up to splice node
3. Reorder points at splice node
4. Resume at next node
5. Repeat 2-4 until finished

Dynamic sorting enhances utilization

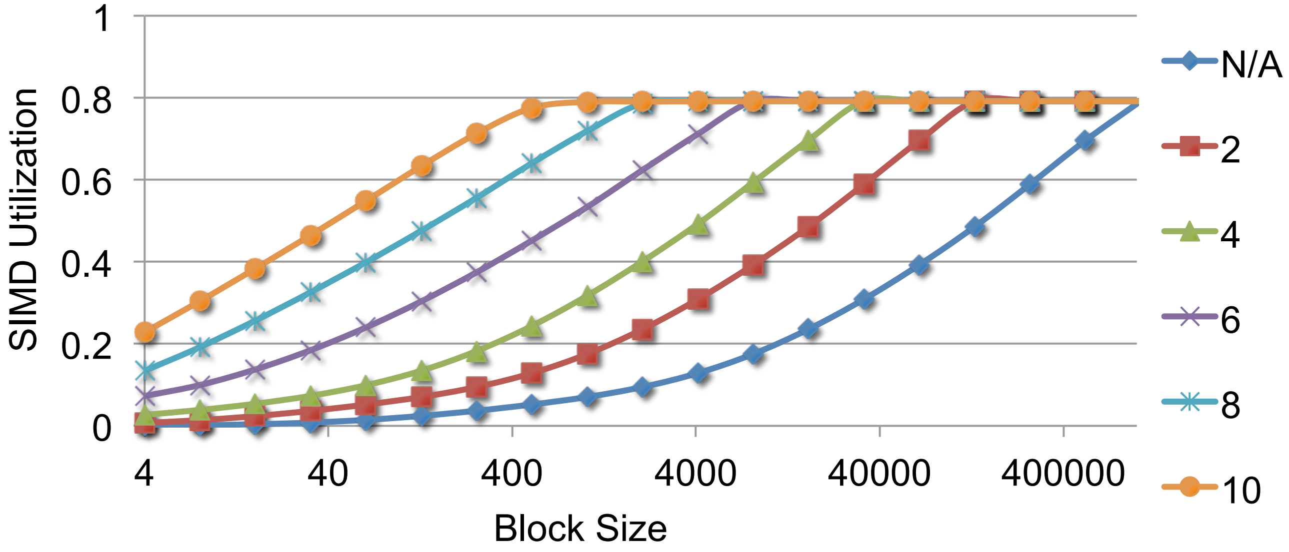


Dynamic sorting enhances utilization



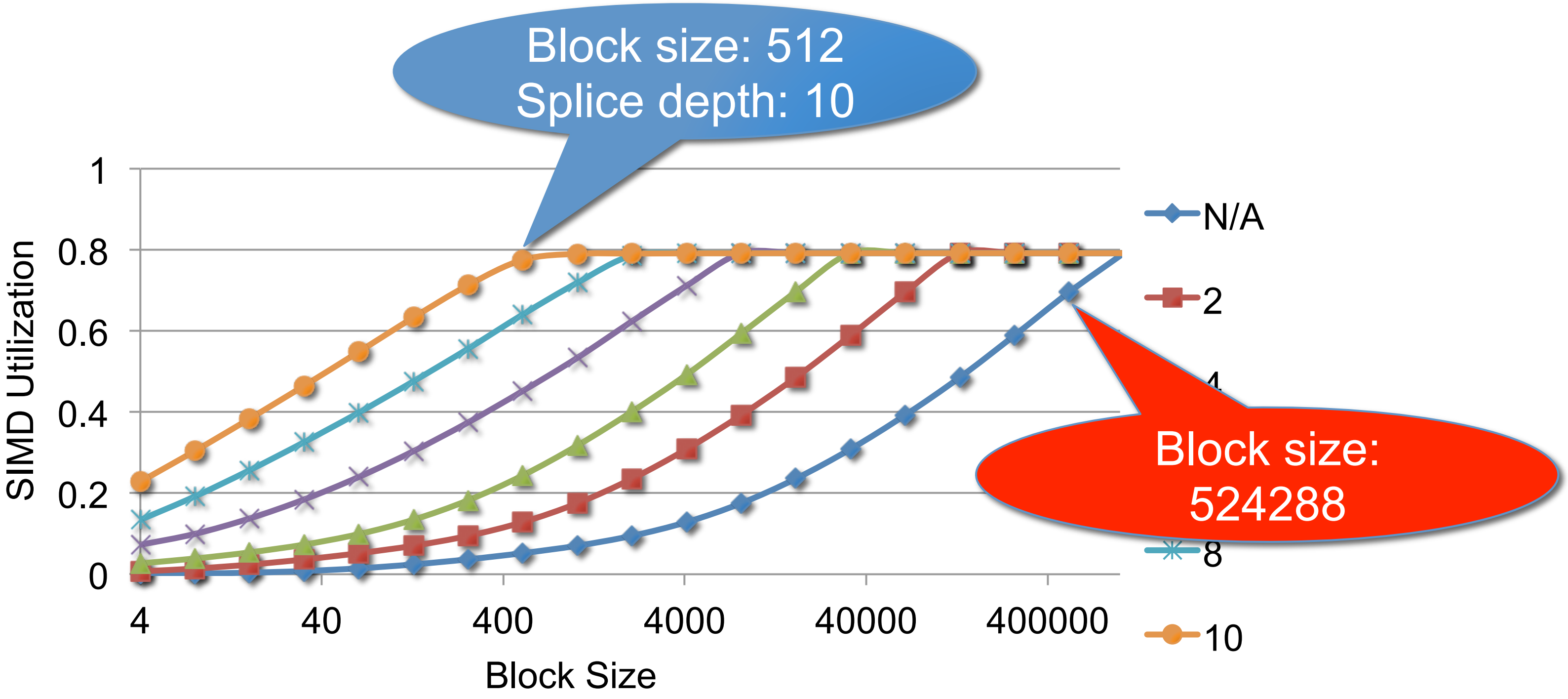
Circles in blue / Total circles = 48 / 74 = 0.65

SIMD utilization – splice depth



Nearest Neighbor

SIMD utilization – splice depth

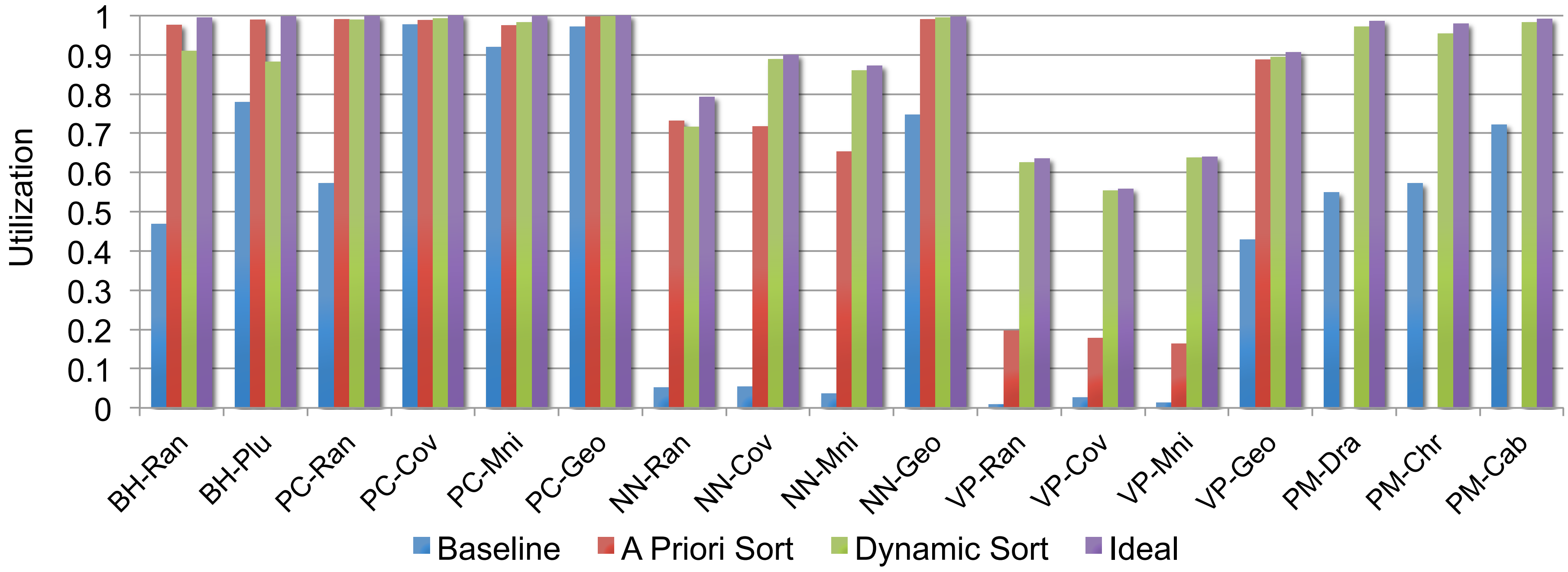


Block size: 512
Splice depth: 10

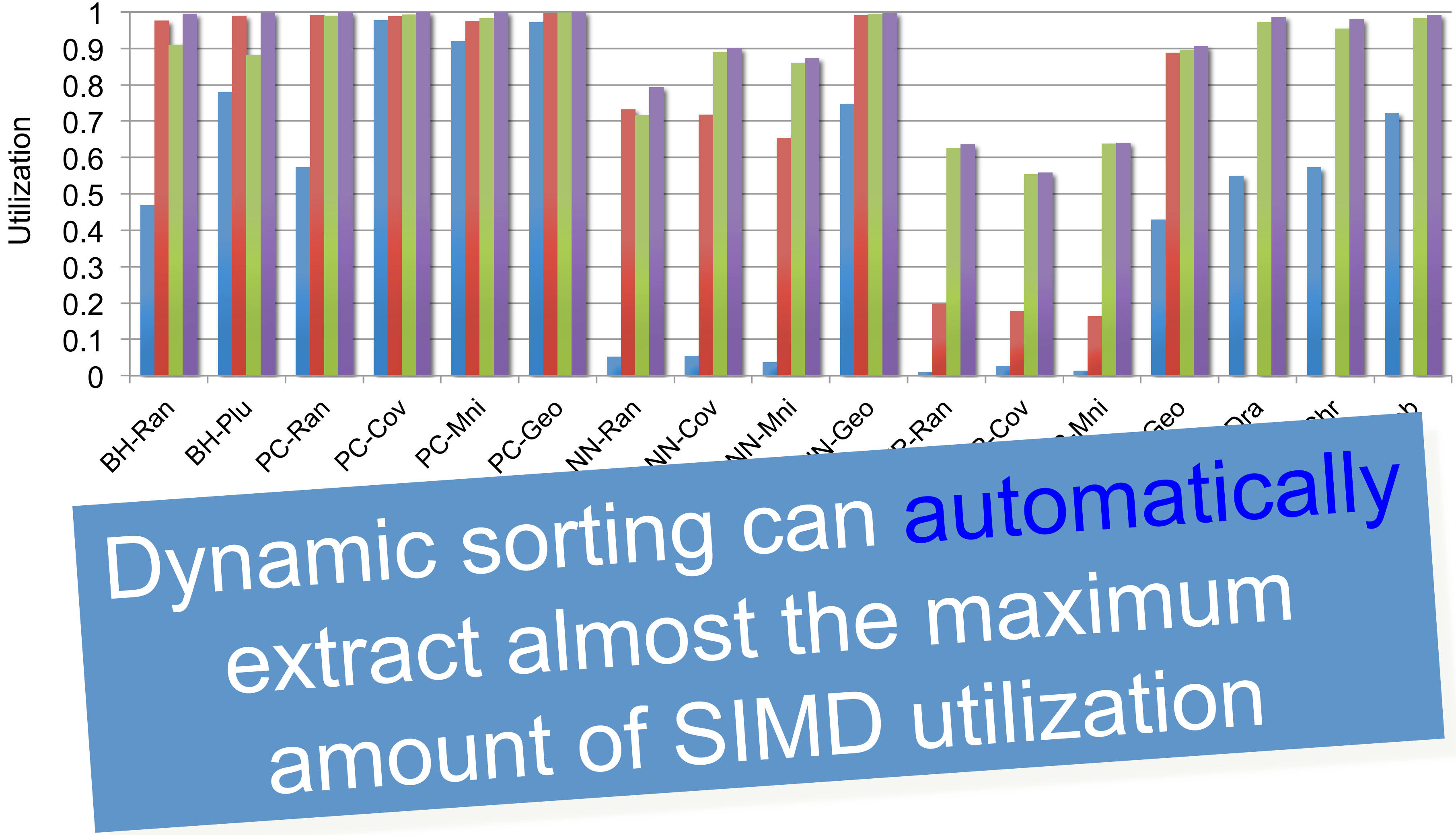
Block size:
524288

Nearest Neighbor

SIMD utilization



SIMD utilization



Dynamic sorting can **automatically** extract almost the maximum amount of SIMD utilization

Outline

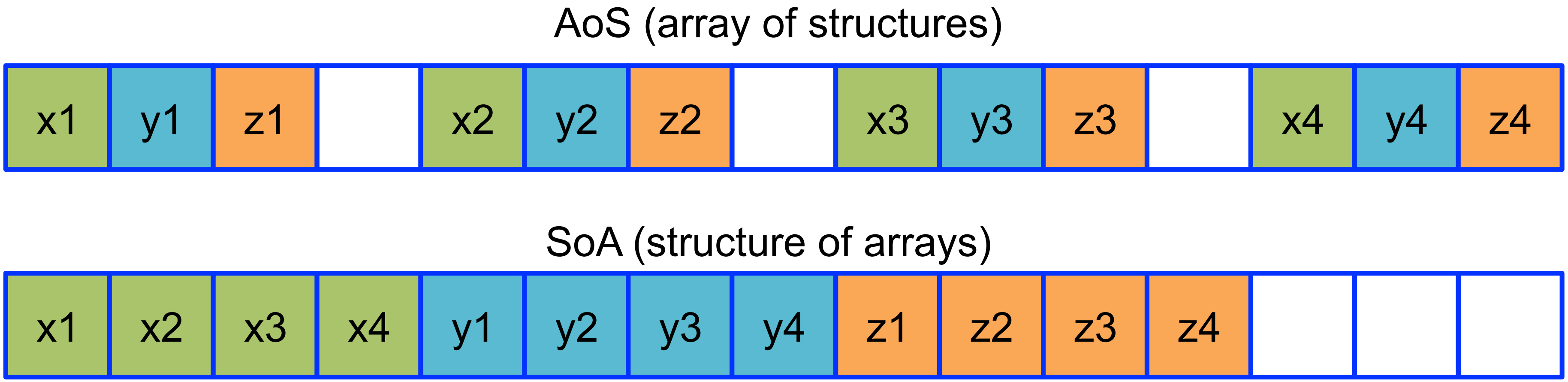
- Example & Abstract Model
- Point Blocking to Enable SIMD
- Traversal Splicing to Enhance Utilization
- Automatic Transformation
- Evaluation and Conclusion

Automatic transformation

- Point blocking
  Jo and Kulkarni [OOPSLA 2011]
- Traversal splicing
  Jo and Kulkarni [OOPSLA 2012]

Automatic transformation

- Our key addition for SIMD:
  Layout transformation from AoS (array of structures) to SoA (structure of arrays)
  - + Allows vector load/stores
  - + Packed data has better spatial locality
  - - More overhead in moving data



AoS (array of structures)

x1	y1	z1		x2	y2	z2		x3	y3	z3		x4	y4	z4

SoA (structure of arrays)

x1	x2	x3	x4	y1	y2	y3	y4	z1	z2	z3	z4			

AoS to SoA layout

- Whole program AoS to SoA layout transformation difficult to automate with aliasing
- Limit scope to traversal code only
  - Copy in to SoA before traversal
  - Copy out to AoS after traversal
- Inter-procedural, flow-insensitive analysis
  - Determine which point fields should be SoA
  - Conservatively ensure correctness

AoS to SoA layout

```
void recurse(Point *p, Node *n) {
  if (truncate(p, n)) return;
  if (n->isLeaf()) {
    update(p, n);
  } else {
    recurse(p, n->left);
    recurse(p, n->right);
  }
}
```

AoS to SoA layout

```
struct Point { float f1, f2, f3; }


void recurse(Point *p, Node *n) {
  if (truncate(p, n)) return;
  if (n->isLeaf()) {
    update(p, n);
  } else {
    recurse(p, n->left);
    recurse(p, n->right);
  }
}
```

AoS to SoA layout

```
struct Point { float f1, f2, f3; }
struct Node { Node *left, *right; Point *point; }

void recurse(Point *p, Node *n) {
  if (truncate(p, n)) return;
  if (n->isLeaf()) {
    update(p, n);
  } else {
    recurse(p, n->left);
    recurse(p, n->right);
  }
}
```

AoS to SoA layout

```
struct Point { float f1, f2, f3; }
struct Node { Node *left, *right; Point *point; }

void recurse(Point *p, Node *n) {
  if (truncate(p, n)) return;
  if (n->isLeaf()) {
    update(p, n);
  } else {
    recurse(p, n->left);
    recurse(p, n->right);
  }
}

bool truncate(Point *p, Node *n) {
  return p->f1 == n->point->f1;
}

void update(Point *p, Node *n) {
  p->f2 += n->point->f3;
}
```

Ensuring correctness

```
struct Point { float f1, f2, f3; }
struct Node { Node *left, *right; Point *point; }

void recurse(Point *p, Node *n) {
  if (truncate(p, n)) return;
  if (n->isLeaf()) {
    update(p, n);
  } else {
    recurse(p, n->left);
    recurse(p, n->right);
  }
}

bool truncate(Point *p, Node *n) {
  return p->f1 == n->point->f1;
}

void update(Point *p, Node *n) {
  p->f2 += n->point->f3;
}
```

Ensuring correctness

```
struct Point { float f1, f2, f3; }
struct Node { Node *left, *right; Point *point; }
```

	Point-access		Non-point-access	
	Read	Write	Read	Write
f1				
f2				
f3				

```
bool truncate(Point *p, Node *n) {
  return p->f1 == n->point->f1;
}

void update(Point *p, Node *n) {
  p->f2 += n->point->f3;
}
```

Ensuring correctness

```
struct Point { float f1, f2, f3; }
struct Node { Node *left, *right; Point *point; }
```

	Point-access		Non-point-access	
	Read	Write	Read	Write
f1	✓			
f2				
f3				

```
bool truncate(Point *p, Node *n) {
  return p->f1 == n->point->f1;
}

void update(Point *p, Node *n) {
  p->f2 += n->point->f3;
}
```

Ensuring correctness

```
struct Point { float f1, f2, f3; }
struct Node { Node *left, *right; Point *point; }
```

	Point-access		Non-point-access	
	Read	Write	Read	Write
f1	✓		✓	
f2				
f3				

```
bool truncate(Point *p, Node *n) {
  return p->f1 == n->point->f1;
}

void update(Point *p, Node *n) {
  p->f2 += n->point->f3;
}
```

Ensuring correctness

```
struct Point { float f1, f2, f3; }
struct Node { Node *left, *right; Point *point; }
```

	Point-access		Non-point-access	
	Read	Write	Read	Write
f1	✓		✓	
f2	✓	✓		
f3				

```
bool truncate(Point *p, Node *n) {
  return p->f1 == n->point->f1;
}

void update(Point *p, Node *n) {
  p->f2 += n->point->f3;
}
```

Ensuring correctness

```
struct Point { float f1, f2, f3; }
struct Node { Node *left, *right; Point *point; }
```

	Point-access		**Non-point-access**	
	Read	Write	Read	Write
f1	✓		✓	
f2	✓	✓		
f3			✓	

```
bool truncate(Point *p, Node *n) {
  return p->f1 == n->point->f1;
}

void update(Point *p, Node *n) {
  p->f2 += n->point->f3;
}
```

Transforming SoA fields

```
struct Point { float f1, f2, f3; }
struct Node { Node *left, *right; Point *point; }
```

	Point-access		**Non-point-access**	
	Read	Write	Read	Write
f1	✓		✓	
f2	✓	✓		
f3			✓	

```
bool truncate(Point *p, Node *n) {
  return p->f1 == n->point->f1;
}

void update(Point *p, Node *n) {
  p->f2 += n->point->f3;
}
```

Transforming SoA fields

```
struct Point { float f1, f2, f3; }
struct Node { Node *left, *right; Point *point; }
```

	Point-access		Non-point-access	
	Read	Write	Read	Write
f1	✓		✓	
f2	✓	✓		
f3			✓	

```
bool truncate(Block *block, int bi, Node *n) {
  return block->f1[bi] == n->point->f1;
}

void update(Block *block, int bi, Node *n) {
  block->f2[bi] += n->point->f3;
}
```

Correctness violation example

```
struct Point { float f1, f2, f3; }
struct Node { Node *left, *right; Point *point; }
```

	Point-access		Non-point-access	
	Read	Write	Read	Write
f1	✓		✓	
f2	✓	✓	✓	
f3				

```
bool truncate(Block *block, int bi, Node *n) {
  return block->f1[bi] == n->point->f1;
}

void update(Block *block, int bi, Node *n) {
  block->f2[bi] += n->point->f2;
}
```

Ensuring correctness

```
struct Point { float f1, f2, f3; }
struct Node { Node *left, *right; Point *point; }
```

	Point-access		Non-point-access	
	Read	Write	Read	Write
f1	✓		✓	
f2	✓	✓	✓	

Sound analysis conservatively proves SoA transformation correct. Suffices to transform all of our benchmarks.

SIMTree

- Implementation of analysis and transformation in a source to source C++ compiler
- Based on ROSE compiler infrastructure
- Transforms code to apply point blocking, traversal splicing, and SoA layout
- Does not perform the vectorization itself
- https://engineering.purdue.edu/plcl/simtree/

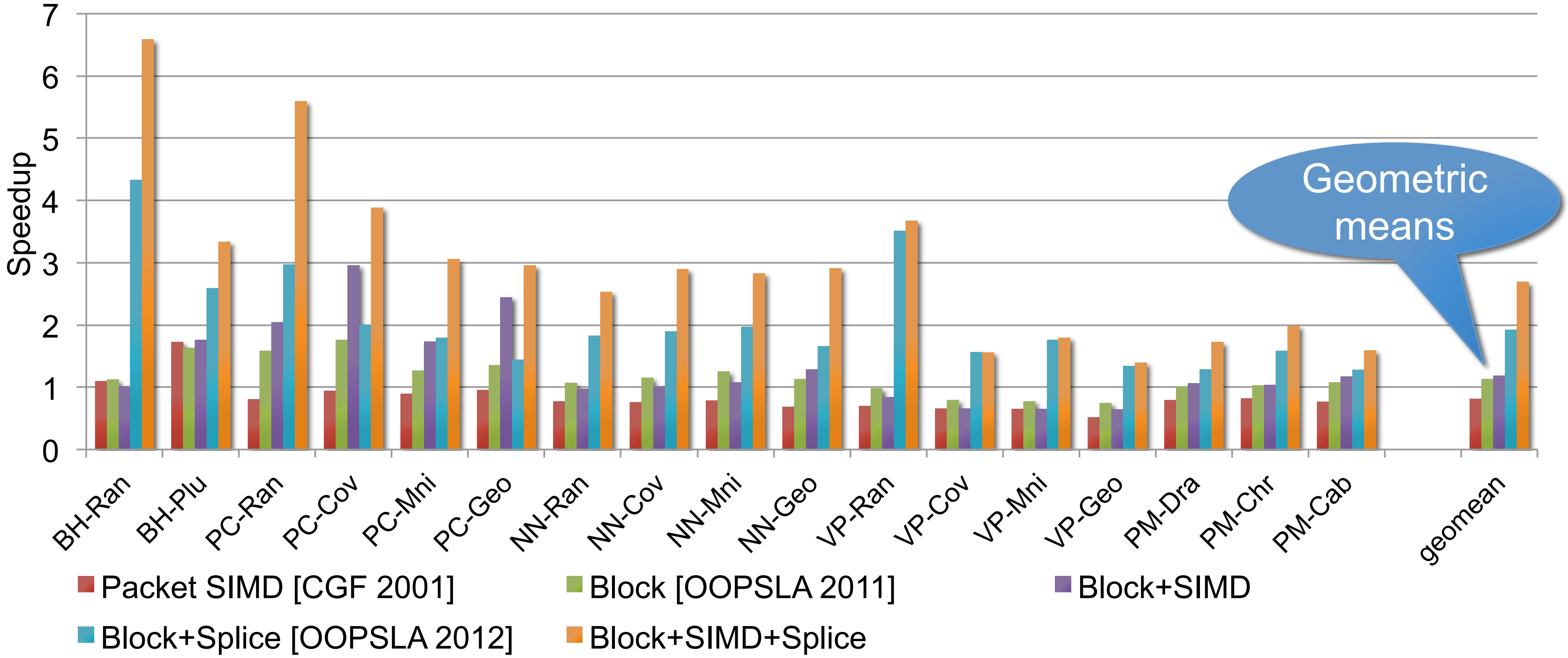
Outline

- Example & Abstract Model
- Point Blocking to Enable SIMD
- Traversal Splicing to Enhance Utilization
- Automatic Transformation
- Evaluation and Conclusion

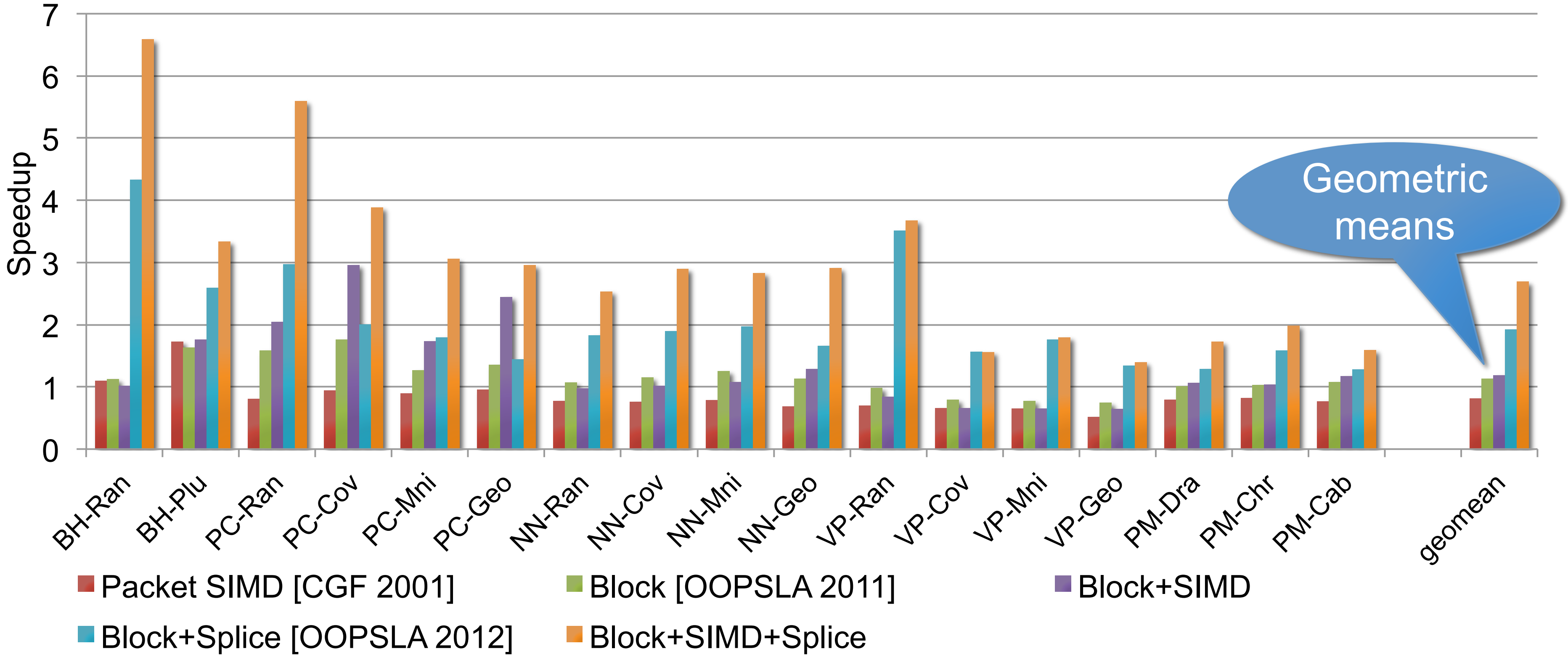
Evaluation

- Five benchmarks
  - Barnes-Hut, Point Correlation, Nearest Neighbor, Vantage Point, Photon Mapping
- Real and random inputs form 17 benchmark/inputs
- Two machines
  - Intel Xeon E5-4650
  - AMD Opteron 6282
- Automatic transformation with SIMTree
- Manual vectorization of transformed code with 4-way SIMD intrinsics for best performance
  - Auto vectorization of transformed code with icc gets 84% of best performance

Speedup on Xeon



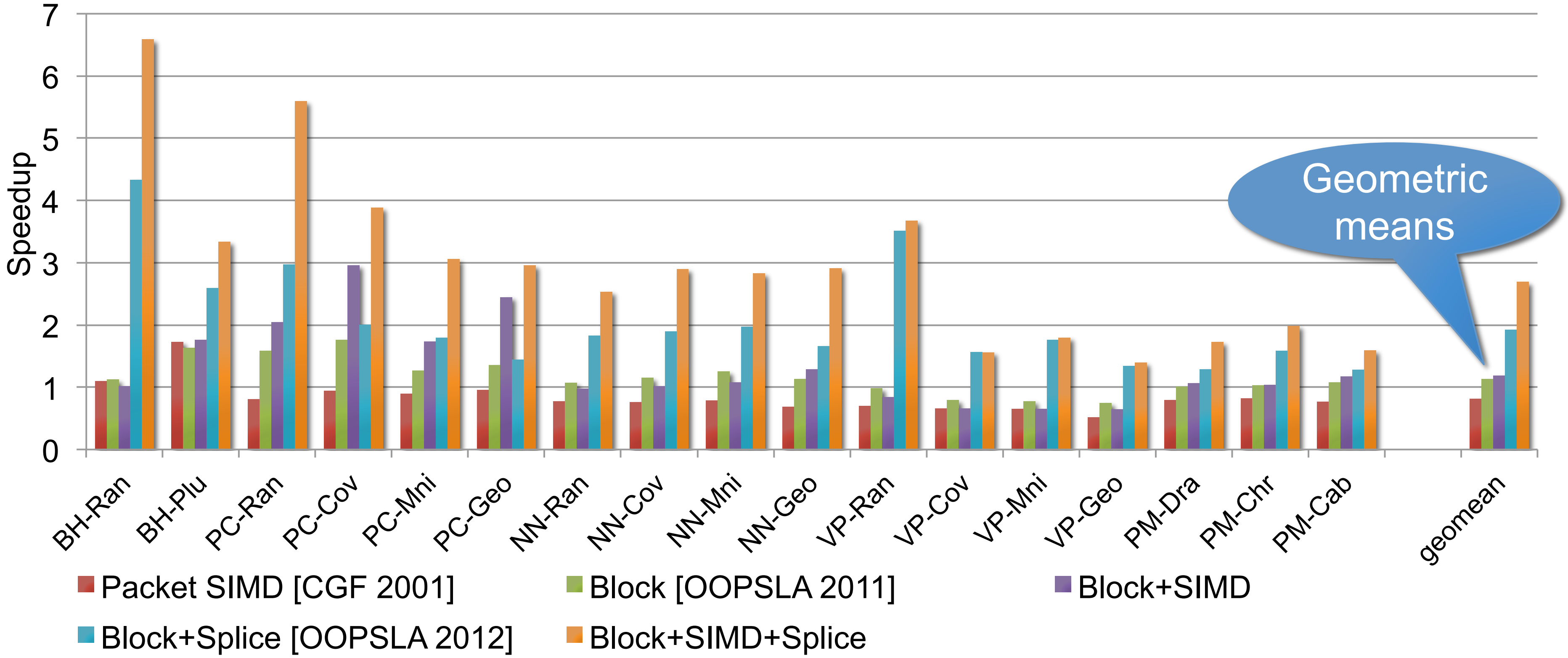
Speedup on Xeon



Geometric means

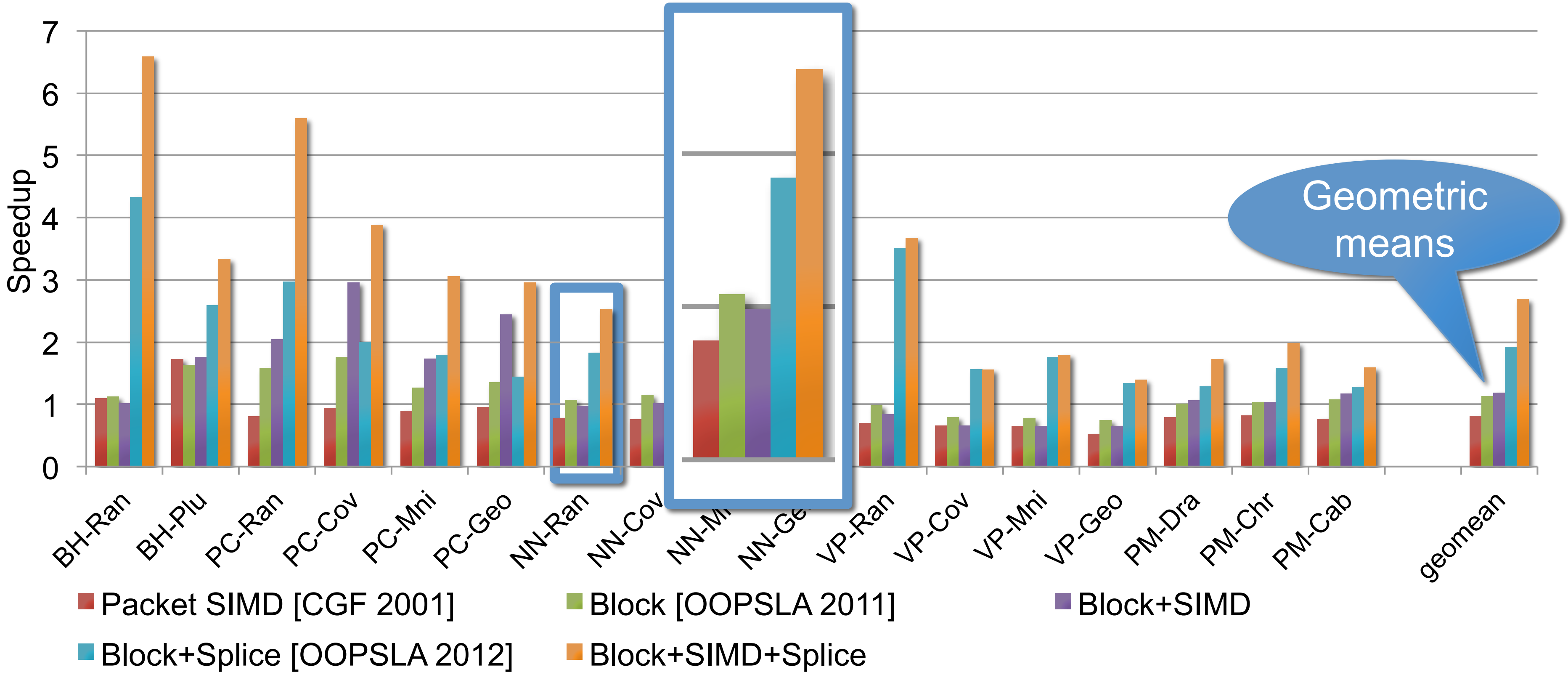
	PacketSIMD	Block	Block +SIMD	Block +Splice	Block+SIMD +Splice
Xeon	0.81	1.13	1.19	1.92	2.69

Dynamic sorting makes SIMD profitable



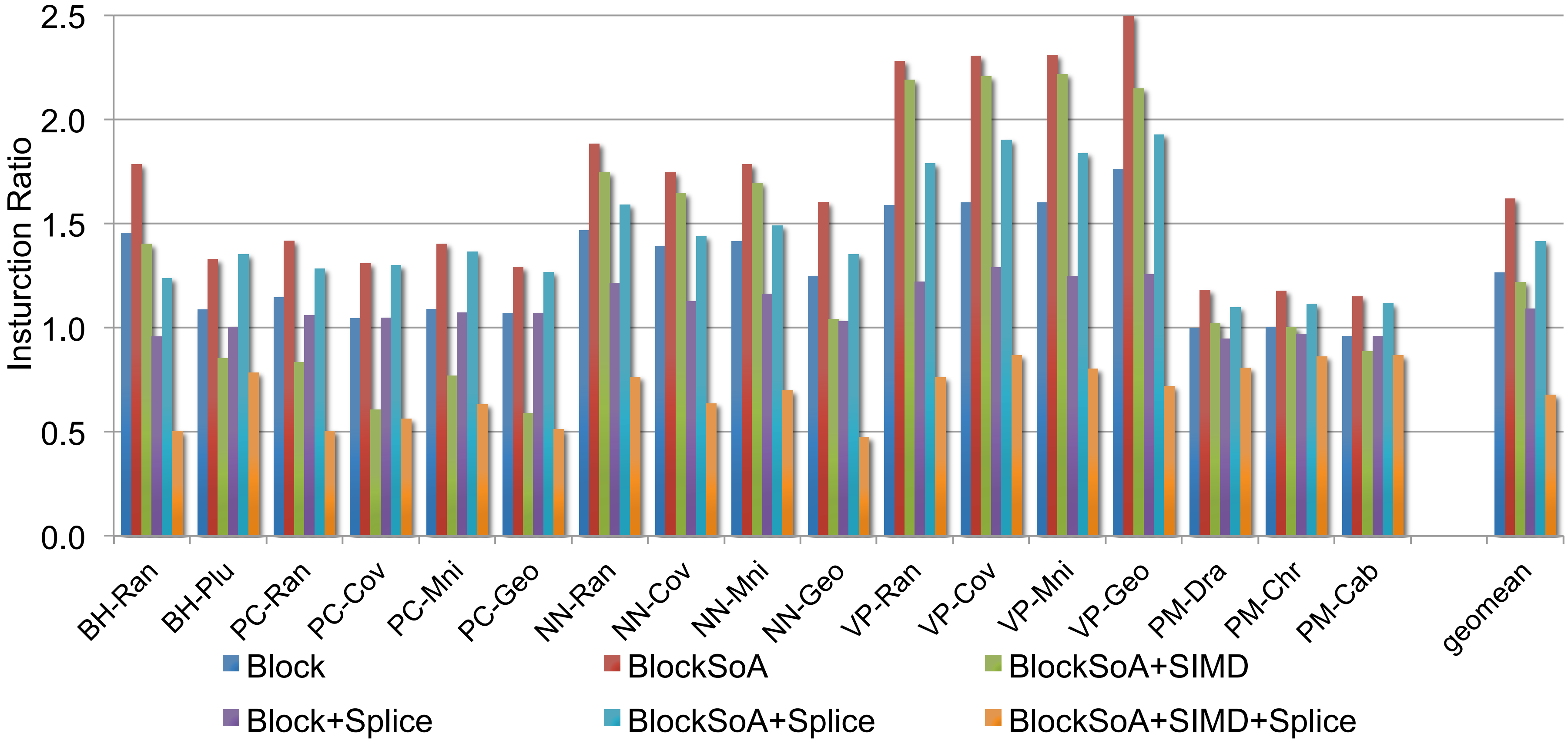
	PacketSIMD	Block	Block +SIMD	Block +Splice	Block+SIMD +Splice
Xeon	0.81	1.13	1.19	1.92	2.69
Opteron	0.83	1.15	1.27	1.78	2.86

Dynamic sorting makes SIMD profitable



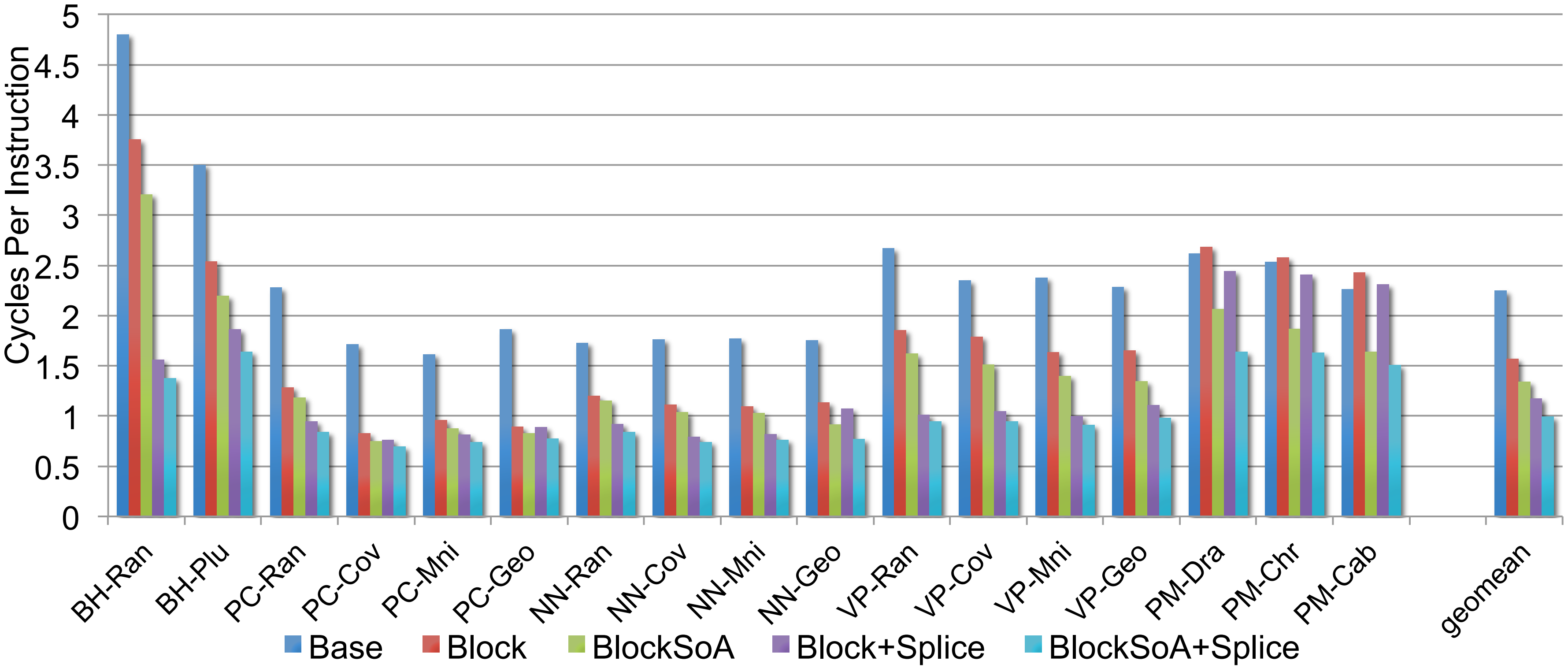
	PacketSIMD	Block	Block +SIMD	Block +Splice	Block+SIMD +Splice
Xeon	0.81	1.13	1.19	1.92	2.69
Opteron	0.83	1.15	1.27	1.78	2.86

Instruction counts: Opteron



Block	BlockSoA	BlockSoA +SIMD	Block +Splice	BlockSoA +Splice	BlockSoA +SIMD+Splice
1.27	1.62	1.20	1.08	1.38	0.64

Cycles per instruction: Opteron



Base	Block	BlockSoA	Block+Splice	BlockSoA+Splice
2.24	1.56	1.35	1.17	1.04

Conclusion

- Show how tree traversal codes can be systematically transformed to
    - Expose SIMD opportunities
    - Enhance utilization
- Propose a novel layout transformation for efficient vectorization of tree codes
- Present a framework for automatically restructuring traversals and data layouts to enable vectorization

Conclusion

- Show how tree traversal codes can be systematically transformed to
  - Expose SIMD opportunities
  - Enhance utilization
- Propose a novel layout transformation for efficient vectorization of tree codes
- Present a framework f

SIMTree is open source!
https://engineering.purdue.edu/plcl/simtree/

AUTOMATIC VECTORIZATION OF TREE TRAVERSALS

Youngjoon Jo, Michael Goldfarb and Milind Kulkarni

Google™ Purdue University

PACT, Edinburgh, U.K.

September 11th, 2013