RICE George R. Brown School of Engineering Computer Science

Analysis and Transformation of Programs with Explicit Parallelism

Vivek Sarkar

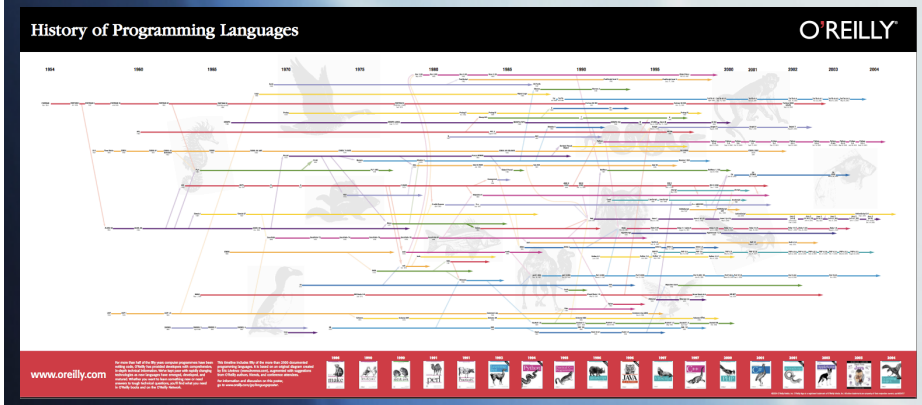
E.D. Butcher Chair in Engineering

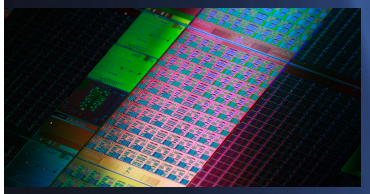
Professor of Computer Science

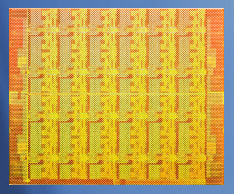
Rice University

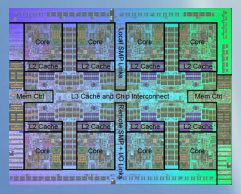
vsarkar@rice.edu

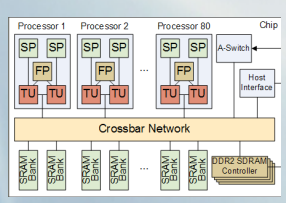
June 30, 2013

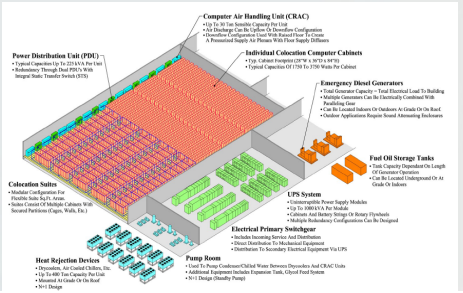












Parallelism is Ubiquitous

Diversity of Parallel Programming Models

- Thread-based libraries
  - Parallelism is exploited via library calls
  - Examples: Pthreads, Intel Threading Building Blocks, Java Concurrency, Microsoft .Net Task Parallel Library
- Directive-based models
  - Simplified pragma syntax for expressing parallelism; for many programs, semantics is preserved if pragmas are elided
  - Example: OpenMP
- Programming languages with explicit parallelism
  - Targets shared and distributed memory systems
  - Examples: Cilk (MIT), Cilk++ (Intel), Unified Parallel C, Co-Array Fortran, CUDA (NVIDIA), OpenCL, Chapel (Cray), X10 (IBM), Habanero-Java (Rice)

Rice Habanero Multicore Software Project: Enabling Technologies for Extreme Scale

Parallel Applications

Portable execution model

1) Lightweight asynchronous tasks and data transfers
- Creation: *async tasks, future tasks, data-driven tasks*
- Termination: *finish, future get, await*
- Data Transfers: *asyncPut, asyncGet, asyncISend, asyncIRecv*

2) Locality control for task and data distribution
- Task Distributions: *hierarchical places*
- Data Distributions: *hierarchical places, global name space*

3) Inter-task synchronization operations
- Mutual exclusion: *isolated, actors*
- Collective and point-to-point synchronization: *phasers*

Habanero Programming Languages

Habanero Static Compiler & Parallel Intermediate Representation

Habanero Runtime System

Two-level programming model

Declarative Coordination Language for Domain Experts,

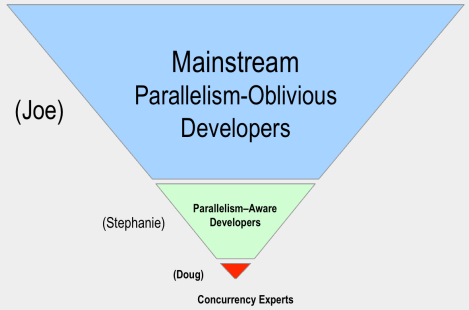
CnC (Intel Concurrent Collections)

+

Task-Parallel Languages for Parallelism-aware Developers:

Habanero-Java (from X10 v1.5),

Habanero-C, Habanero-Scala



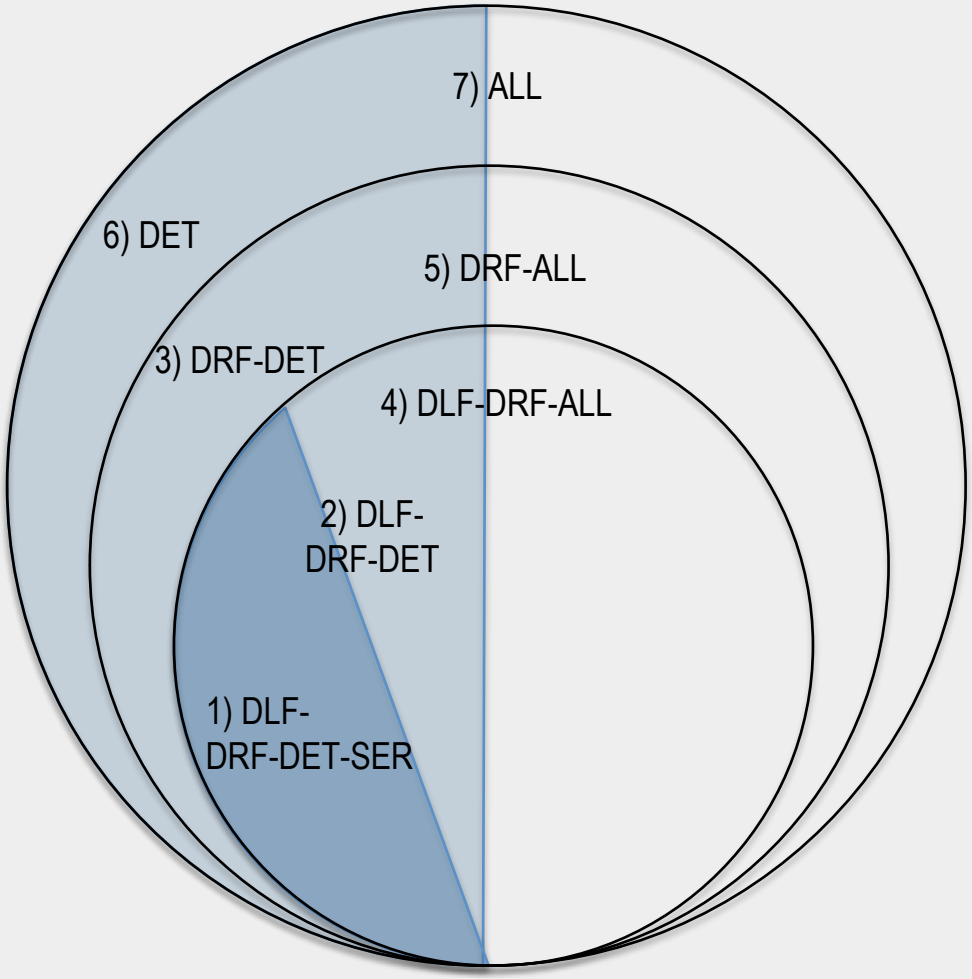
Extreme Scale Platforms

Elements of Habanero Execution Model

1) Lightweight asynchronous tasks and data transfers
   - Creation: *async tasks, future tasks, data-driven tasks*
   - Termination: *finish, future get, await*
   - Data Transfers: *asyncPut, asyncGet, asyncISend, asyncIRecv*
2) Locality control for task and data distribution
   - Task Distributions: *hierarchical places*
   - Data Distributions: *hierarchical places, global name space*
3) Inter-task synchronization operations
   - Mutual exclusion: *global/object-based isolation, actors*
   - Collective and point-to-point synchronization: *phasers*

Goal: unified model of parallelism that spans programming models, compilers, runtime systems, and provides a pedagogic foundation for teaching

Classification of Habanero Parallel Programs



- Legend
  - DLF = DeadLock-Free
  - DRF = Data-Race-Free
  - DET = Determinate
  - DRF➔DET = DRF implies DET
  - SER = Serializable
- *If a Habanero program only uses async, finish, and future constructs (no mutual exclusion), then it is guaranteed to belong to the DLF + DRF➔DET + SER class*
- *Adding phasers yields programs in the DLF + DRF➔DET class*
- *Adding async await yields programs in the DLF + DRF➔DET class*
- *Restricting shared data accesses to futures, isolated, actors yields programs in the DRF-ALL class*

"Habanero-Java: the New Adventures of Old X10." Vincent Cave, Jisheng Zhao, Jun Shirako, Vivek Sarkar PPPJ 2011.

Effectiveness of Data Race Detection depends on Execution Model primitives

Properties	OTFDAA [PLDI '89]	Offset-Span [SC '91]	SP-bags [SPAA '97]	SP-hybrid [SPAA '04]	FastTrack [PLDI '09]	ESP-bags [RV '10]	SPD3 [PLDI '12]
Target Language	Nested Fork-Join & Synchronization operations	Nested Fork-Join	Spawn-Sync	Spawn-Sync	Unstructured Fork-Join	Structured Async-Finish	Structured Async-Finish
Space Overhead per memory location	O(m)	O(1)	O(1)	O(1)	O(N)	O(1)	O(1)
Guarantees	Per-Schedule	Per-Input	Per-Input	Per-Input	Per-Input	Per-Input	Per-Input
Empirical Evaluation	No	Minimal	Yes	No	Yes	Yes	Yes
Execute Program in Parallel	Yes	Yes	No	Yes	Yes	No	Yes
Dependent on Scheduling technique	No	No	Yes	Yes	No	Yes	No

"Scalable and Precise Dynamic Data Race Detection for Structured Parallelism." Raghavan Raman, Jisheng Zhao, Vivek Sarkar, Martin Vechev, Eran Yahav. PLDI 2012.

RICE

Target Platforms

Habanero programs have been executed on a wide range of production and experimental systems

- Multicore SMPs (AMD, IBM, Intel)
- Discrete GPUs (AMD, NVIDIA)
- Integrated GPUs (AMD, Intel)
- FPGA (Convey, w/ GPU added)
- Clusters
- Cyclops
- SCC
- . . .

Pedagogy using Habanero execution model, COMP 322: Fundamentals of Parallel Programming

- **Sophomore-level CS Course at Rice**
  - https://wiki.rice.edu/confluence/display/PARPROG/COMP322
  - Or do a web search on "comp322 wiki"
- **Approach – mid-level parallel programming model**
  - **"Simple things should be simple, complex things should be possible"**
  - Introduce students to fundamentals of parallel programming
    - Primitive constructs for task creation & termination, collective & point-to-point synchronization, task and data distribution, and data parallelism
    - Abstract models of parallel computations and computation graphs
    - Parallel algorithms & data structures including lists, trees, graphs, matrices
    - Common parallel programming patterns
  - Use Habanero-Java (HJ) as pedagogic language to understand fundamentals for two-thirds of course, and then teach standard parallel programming models (Java threads, MPI, CUDA) using HJ principles

Rice Habanero Multicore Software Project: Enabling Technologies for Extreme Scale

Parallel Applications

Portable execution model

1) Lightweight asynchronous tasks and data transfers
- Creation: *async tasks, future tasks, data-driven tasks*
- Termination: *finish, future get, await*
- Data Transfers: *asyncPut, asyncGet, asyncISend, asyncIRecv*

2) Locality control for task and data distribution
- Task Distributions: *hierarchical places*
- Data Distributions: *hierarchical places, global name space*

3) Inter-task synchronization operations
- Mutual exclusion: *isolated, actors*
- Collective and point-to-point synchronization: *phasers*

Habanero Programming Languages

Habanero Static Compiler & Parallel Intermediate Representation

Habanero Runtime System

Two-level programming model

Declarative Coordination Language for Domain Experts,

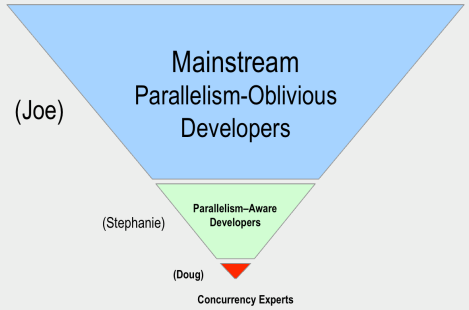
CnC (Intel Concurrent Collections)

+

Task-Parallel Languages for Parallelism-aware Developers:

Habanero-Java (from X10 v1.5),

Habanero-C, Habanero-Scala



Extreme Scale Platforms

Two approaches to compiling for parallelism

1. Automatic extraction of parallelism from sequential programs
   - Past work from the last 30+ years has led to fairly mature compiler technologies in this area
   - New hardware platforms continue to provide new challenges
2. Compilation and optimization of explicitly parallel programs
   - Increase in languages with explicit parallelism (as evidenced by this workshop)
   - New compiler foundations needed for programs with explicit parallelism

➔ In general, we need a combination of 1. and 2.

Three Levels of Parallel Intermediate Representations

- High-level PIR (HPIR)
  - Retain high-level loop constructs
  - Retain hierarchical structure of parallelism in a Program Structure Tree (PST)
- Middle-level PIR (MPIR)
  - Flatten control flow
  - Convert to lower-level parallel constructs (async, finish)
- Low-level (PIR)
  - Include code generation for target runtime system

Motivation: compiler optimizations can be performed at all three levels

Related References

[BZS13] Interprocedural Strength Reduction of Critical Sections in Explicitly-Parallel Programs. Rajkishore Barik, Jisheng Zhao, Vivek Sarkar. The 22nd International Conference on Parallel Architectures and Compilation Techniques (PACT), September 2013 (to appear).

[NSZS13] A Transformation Framework for Optimizing Task-Parallelism Programs. V. Krishna Nandivada, Jun Shirako, Jisheng Zhao, Vivek Sarkar. ACM Transactions on Programming Languages and Systems (TOPLAS), Volume 35, May 2013.

[BZGPBS10] Communication Optimizations for Distributed-Memory X10 Programs. Rajkishore Barik, Jisheng Zhao, David Grove, Igor Peshansky, Zoran Budimlić, Vivek Sarkar. 25th IEEE International Parallel and Distributed Processing Symposium (IPDPS), April 2011.

[ZSNS10] Reducing Task Creation and Termination Overhead in Explicitly Parallel Programs. Jisheng Zhao, Jun Shirako, Krishna Nandivada, Vivek Sarkar. The Nineteenth International Conference on Parallel Architectures and Compilation Techniques (PACT), September 2010.

[BS09] Interprocedural Load Elimination for Dynamic Optimization of Parallel Programs. Rajkishore Barik, Vivek Sarkar. The Eighteenth International Conference on Parallel Architectures and Compilation Techniques (PACT), September 2009.

[SZNS09] Chunking Parallel Loops in the Presence of Synchronization. Jun Shirako, Jisheng Zhao, Krishna Nandivada, Vivek Sarkar. Proceedings of the 2009 ACM International Conference on Supercomputing (ICS), June 2009.

Outline of Today's Lecture

- HPIR Example: A Transformation Framework for Optimizing Task-Parallel Programs [NSZS13]

- MPIR Example: Load Elimination [BS09]

Recap: async & finish constructs in X10 & Habanero-Java

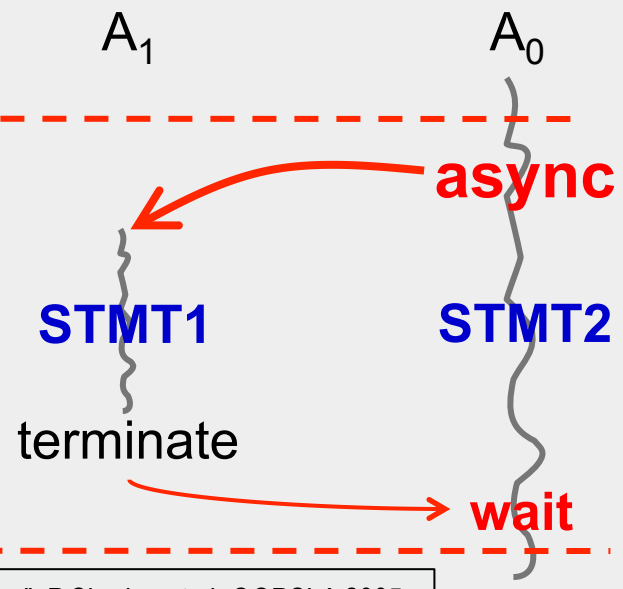
async S

- Creates a new child task that executes statement S
  - Like OpenMP's task pragma
- Parent task moves on to statement following the async
- Can be used to implement higher level constructs like forall loops

finish S

- Execute S, but wait until *all* (transitively) spawned asyncs in S's scope have terminated
  - Like OpenMP's taskwait
- Implicit finish between start and end of main program
- Use of finish cannot create a deadlock cycle

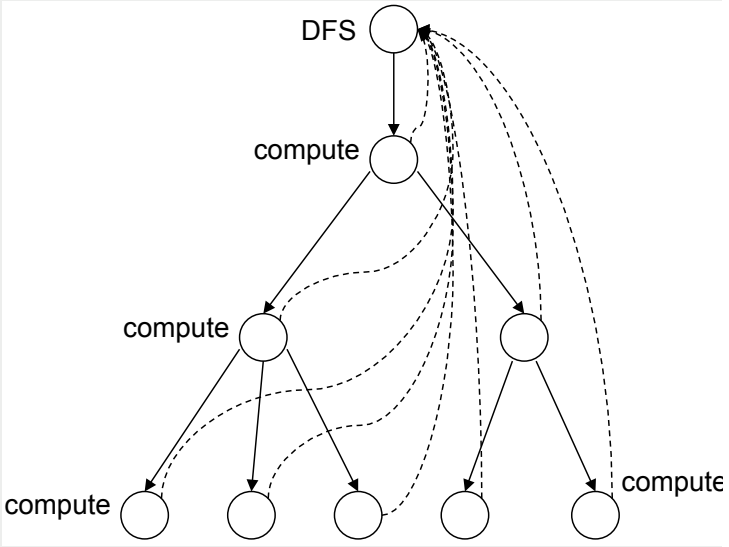
```
//A0 (Parent)
finish {    //Begin finish
  async {
    STMT1;  //A1 (Child)
  }
  STMT2;    //A0
}           //End finish
```



"X10: An Object-oriented approach to non-uniform Clustered Computing", P.Charles et al. OOPSLA 2005.

Parallel Spanning Tree Algorithm in Habanero-Java

```
1. class V  {
2.   V [] neighbors; // Input adjacency list
3.   V parent; // Output spanning tree
4.   . . .
5.   boolean tryLabeling(V n) {
6.     boolean retVal = false;
7.     isolated(this) // Object-based isolation
8.       if (parent == null) {
9.         parent = n; retVal = true; }
10.    return retVal;
11.  } // tryLabeling
12.  void compute() {
13.    for (int i=0; i<neighbors.length; i++) {
14.      V child = neighbors[i];
15.      if (child.tryLabeling(this))
16.          async child.compute(); //escaping async
17.     }
18.  } // compute
19. } // class V
20. root.parent = root; //Use self-cycle to identify root
21. finish root.compute();
```



HJ's forall statement = finish + for + async + barriers

Goal 1 (minor): replace common finish-forasync idiom by forall e.g., replace

```
finish for(point [I,J] : [0:N-1,0:N-1]) async
  for (point[K] : [0:N-1])
    C[I][J] += A[I][K] * B[K][J];
```

by

```
forall (point [I,J] : [0:N-1,0:N-1])
  for (point[K] : [0:N-1])
    C[I][J] += A[I][K] * B[K][J];
```

Goal 2 (major): Also support barrier synchronization (next), with extension for next-single statements

OpenMP does not allow barriers in parallel loops (non-conformable example)

```
omp_set_num_threads(m); // m = "number of hardware threads"
delta = epsilon+1; iters = 0;
#pragma omp parallel for
for (int j = 1 ; j <= n ; j++ ) {
    body(…);
}
void body(…) {
    while ( delta > epsilon ) {
        newA[j] = (oldA[j-1]+oldA[j+1])/2.0 ;
        diff[j] = abs(newA[j]-oldA[j]);
        #pragma omp barrier
        if (j == 1) {
            delta = sum(diff); iters++;
            temp = newA; newA = oldA; oldA = temp;
        }
        #pragma omp barrier
} }
```

Unpredictable results on different platforms

Compile-time error, runtime error, deadlock, correct execution if n = m, …

Barrier Synchronization: HJ's "next" statement in forall construct

```
1. forall (point[i] : [0:m-1]) {
2.   int sq = i*i;
3.   System.out.println("Hello from task with sq = " + sq);
4.   next;
5.   System.out.println("Goodbye from task with sq = " + sq);
6. }
```

Phase 0

Phase 1

- next ➔ each forall iteration suspends at next until all iterations arrive (complete previous phase), after which the phase can be advanced
  - If a forall iteration terminates before executing "next", then the other iterations do not wait for it
  - Scope of synchronization is the closest enclosing forall statement
  - Special case of "phaser" construct

next-with-single statement (extension of barrier)

- Goal: rewrite Hello-Goodbye example so as to print a single log message in between phases
- Solution: use next-with-single-statement

```
1. forall (point[i] : [0:m-1]) {
2.   int sq = i*i;
3.   System.out.println("Hello from task with sq = " + sq);
4.   next // next-with-single statement
5.     System.out.println("LOG: Between Hello & Goodbye phases");
6.   System.out.println("Goodbye from task with sq = " + sq);
7. }
```


Legality of Loop Distribution for a Sequential Program

Is it legal to distribute the following loop, assuming that f() and g() are unanalyzable functions?

```
for (int i = ...) {
  /* S1 */   X[f(i)] = ... ;
  /* S2 */   ... = X[g(i)];
}
```


Legality of Loop Distribution for a Parallel Program

Is it legal to distribute the following loops, assuming that f() and g() are unanalyzable functions?

```
for (int i = ...) {//Loop 1
  /* S1 */  X[f(i)] = ... ;
  /* S2 */  async ... = X[g(i)];
}

forall (point[i] : ...) {//Loop 2
  /* S1 */  X[f(i)] = ... ;
  /* S2 */  ... = X[g(i)];
}
```

We need a precise definition of data dependence in parallel programs to answer this question

This is a fundamental question for compiler transformations and for program refactorings

Dynamic Happens-Before (HB) Relation in Task Parallel Programs

The relation HB on instances I_A and I_B of statements A and B is the smallest relation satisfying the following conditions

- (Sequential order) If I_A and I_B belong to the same task, and I_B is sequentially control or data dependent on I_A, then $HB(I_A,I_B)$ = true.
- (Async creation) If I_A is an instance of an async statement, and I_B is the corresponding instance of the first statement in the body of the async, then $HB(I_A,I_B)$ = true.
- (Finish termination) If I_A is the last statement of an async task, and I_B is the end-finish statement instance of I_A's immediately-enclosing-finish (IEF) instance, then $HB(I_A,I_B)$ = true.
- (Transitivity) If $HB(I_A,I_B)$ = true and $HB(I_B,I_C)$ = true then $HB(I_A,I_C)$ = true.

Static Happens-Before Dependence (HBD) Relation

- We say that HBD(A, B) = true if there is a possible execution of the program with instances I_A and I_B of statements A and B that satisfies all the following conditions:
  - (1) $HB(I_A,I_B)$ = true,
  - (2) I_A and I_B access the same location X and at least one of the accesses is a write, and
  - (3) There is no statement instance I_C that writes X such that $HB(I_A,I_C)$ = true and $HB(I_C,I_B)$ = true.
- As with dependence analysis of sequential programs, we classify the dependence as flow, anti, and output when the accesses performed by I_A and I_B are read-after-write, write-after-read, and write-after-write respectively.
- HBD is a "may dependence" analysis (conservative)
- HBD relation can be qualified by restricting the sets of instances participating in the dependence e.g., using direction vectors and distance vectors
- HBD relation degenerates to sequential data dependences when the input program is sequential.

Extending traditional loop transformations for task parallel programs [NSZS13, Fig 9]

1. Serial loop distribution:

```
for (...) { S1;S2; }
// no dependence cycle between S1 & S2
```
⟹
```
for (...) {S1;}
for (...) {S2;}
```

2. Parallel loop distribution:

```
forall (point p : R1)
  { S1; S2; }
// S1 has no dependence on S2
```
⟹
```
forall (point p : R1) S1;
forall (point p : R1) S2;
```

3. Loop/Finish interchange:

```
for (S1;cond;S2)
  finish S3;
// Say E_s = set of e-asyncs in S3
// ¬∃e ∈ E_s: cond has dependence on e
// ¬∃e ∈ E_s:body of e has loop
//      carried dependence on S2, cond or S3
```
⟹
```
S1;
finish
  for (;cond;S2)
    S3;
```

4. Serial-parallel loop interchange:

```
for (i: [1..n])
  forall (point p : R1)   S;
// iterations of the for loop are independent.
// R1 does not depend on i
```
⟹
```
forall (point p : R1)
  for (i: [1..n])
    S;
```

5. Parallel-serial loop interchange:

```
forall (point p : R1)
  for (point q : R2)  S
// R2 is independent of p
// S contains no break/continue
```
⟹
```
for (point q : R2)
  forall (point p : R1)
    S
```

NOTE: these rules are also extended in the paper for precise exception semantics

Extending traditional loop transformations for task parallel programs [NSZS13, Fig 9] (contd)

6. Loop unpeeling:

```
forall (point p: R) S1;
S2;
// no break/continue in S2.
// Say E_s = set of e-asyncs in S1
// ¬∃e ∈ E_s: S2 has dependence on e
```

⟹

```
forall (point p: R)
  {S1; next S2;}
```

7. Loop fusion:

```
forall (point p: R1) S1;
forall (point p: R2) S2;
// Say E_s = set of e-asyncs in S1
// ¬∃e ∈ E_s: S2 has dependence on e
```

⟹

```
forall (point p: R1||R2)
 {if (R1.contains (p)) S1;
  next;
  if (R2.contains (p)) S2;}
```

8. Loop switching:

```
if (c)
  forall (point p: R)
    S;
```

⟹

```
final boolean v = c;
forall (point p: R)
  if (v) S;
```

9. Parallel loop unswitching:

```
forall (point p : R1)
  if (e) S
//e is a pure function and is independent of p
```

⟹

```
if (e)
  forall (point p : R1) S
```

10. Serial loop unswitching:

```
  for(S2;cond1;S3){
    if (cond2) S4; else S5;
  }
// cond2 has no dependence
//     on S2,S3,S4 and S5,
// cond2 has no side effects
```

⟹

```
if (cond2) {
  for(S2;cond1;S3) S4;
} else {
  for(S2;cond1;S3) S5;
}
```

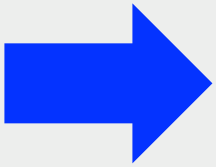

Three example optimizations for parallel tasks and parallel loops

1. Loop chunking
   - chunking fine-grain parallel loops into coarse-grained parallel tasks eliminates the significant overhead for task spawning and scheduling.
2. Forall coarsening
   - reduce task creation and termination overheads by increasing the scope of forall loops
     - *Simple forall-coarsening* increases the granularity of synchronization-free parallelism
     - *Forall-coarsening with synchronization* further increases the granularity of parallelism by adding synchronization operations (SPMDization)
3. Finish elimination
   - eliminate and/or reshape finish regions to reduce synchronization overhead and increase parallelism

Example of Illegal Forall Chunking

```
1:  delta = epsilon+1; iters = 0;
2:  forall (j : [1:n]) {
3:    while (delta > epsilon) {
4:       newA[j] = (oldA[j-1]+oldA[j+1])/2.0;
5:       diff[j] = Math.abs(newA[j]-oldA[j]);
         // sum and exchange
6:       next single {
7:         delta = diff.sum(); iters++;
8:         temp=newA; newA=oldA; oldA=temp;
         } // next single
    } // while
  } // forall
```

Naïve chunking of forall is illegal (iteration j executes multiple while-loop iterations before iteration j+1 starts)

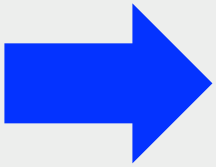


```
1:   delta = epsilon+1; iters = 0;
2:   phaser ph = new phaser(single);
3:   forall ( point[jj] : [1:n:S] ) phased(single(ph)) {
4:     for (int j = jj ; j <= min(jj+S-1,n) ; j++) {
5:       while ( delta > epsilon ) {
6:         newA[j] = (oldA[j-1]+oldA[j+1])/2.0 ;
7:         diff[j] = Math.abs(newA[j]-oldA[j]);
8:         next single { // barrier with single statement
9:           delta = diff.sum(); iters++;
10:          temp = newA; newA = oldA; oldA = temp;
           } // next single
       } // while
     } // for
   } // finish
```

Example of Legal Forall Chunking

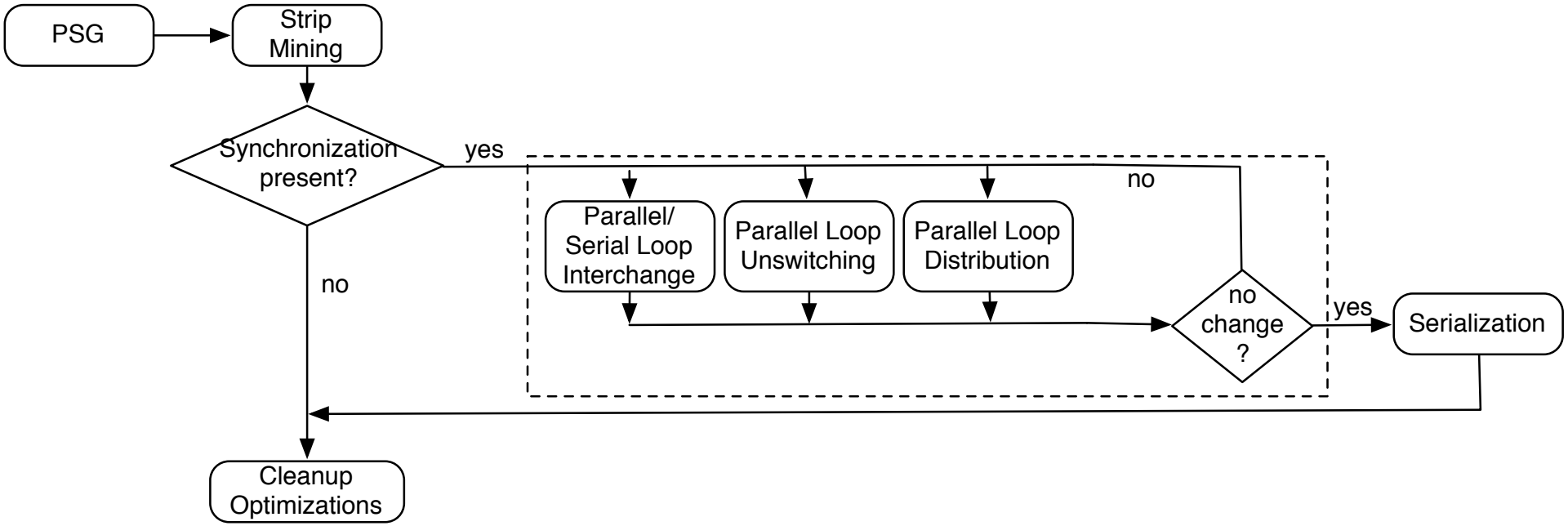
```
1:  delta = epsilon+1; iters = 0;
2:  forall (j : [1:n]) {
3:    while (delta > epsilon) {
4:       newA[j] = (oldA[j-1]+oldA[j+1])/2.0;
5:       diff[j] = Math.abs(newA[j]-oldA[j]);
         // sum and exchange
6:       next single {
7:         delta = diff.sum(); iters++;
8:         temp=newA; newA=oldA; oldA=temp;
         } // next single
    } // while
  } // forall
```

Moving sequential (chunked) j-loop inside while-loop leads to a correct transformation



```
1:   delta = epsilon+1; iters = 0;
2:   phaser ph = new phaser(single);
3:   forall ( point[jj] : [1:n:S] ) phased(single(ph)) {
4:     while ( delta > epsilon ) {
5:       for (int j = jj ; j <= min(jj+S-1,n) ; j++) {
6:         newA[j] = (oldA[j-1]+oldA[j+1])/2.0 ;
7:         diff[j] = Math.abs(newA[j]-oldA[j]);
         } // for
8:       next single { // barrier with single statement
9:         delta = diff.sum(); iters++;
10:        temp = newA; newA = oldA; oldA = temp;
         } // next single
       } // while
     } // finish
```


Loop Chunking Framework



Goal:

Correct chunking transformation to keep original semantics

Step 1: Strip mining (generated nested parallel loops)

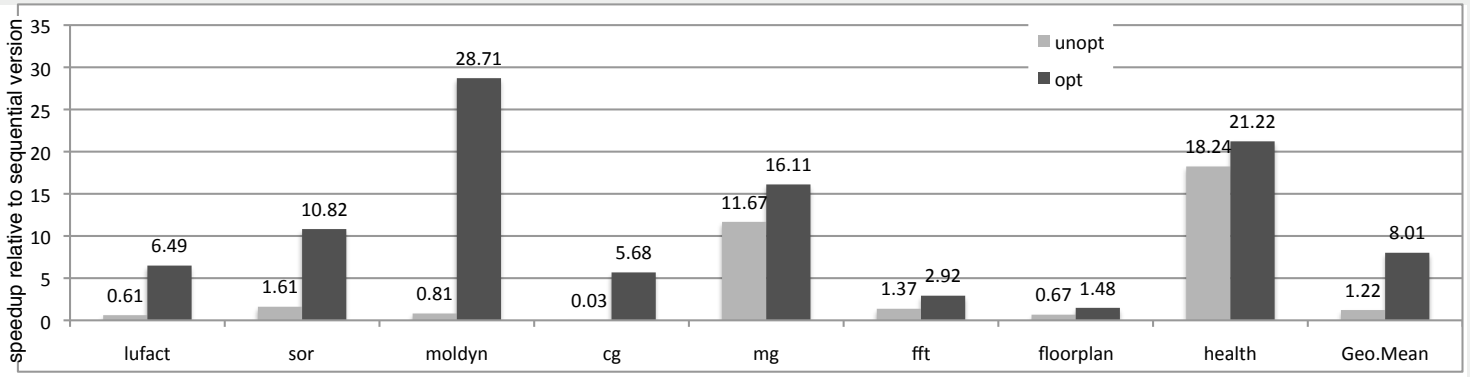
Step 2: Isolation of *next (*combinations of interchange, unswitching, distribution)

Step 3: Serialization of inner strip-mined parallel loop

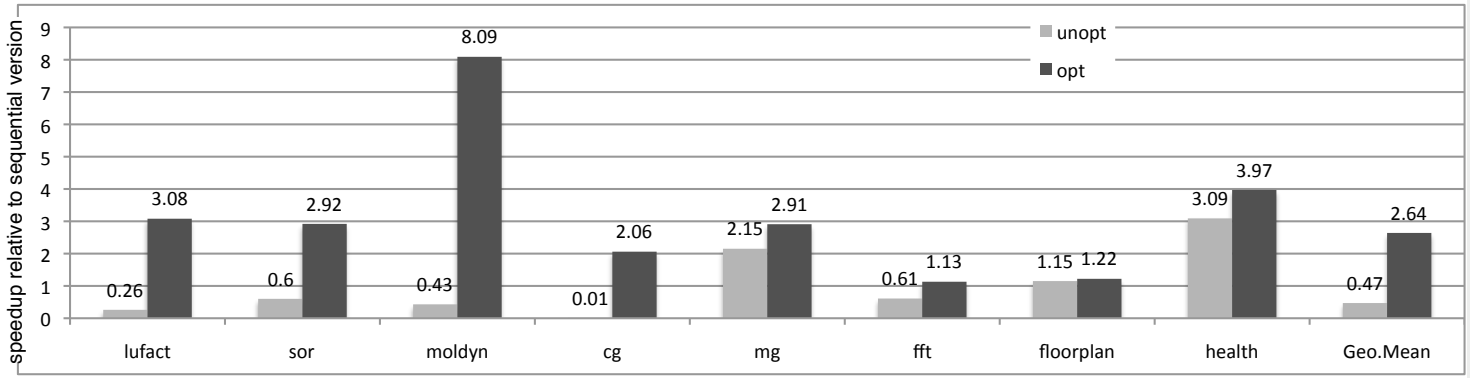
Performance Results for Loop Chunking

unopt =
- original code

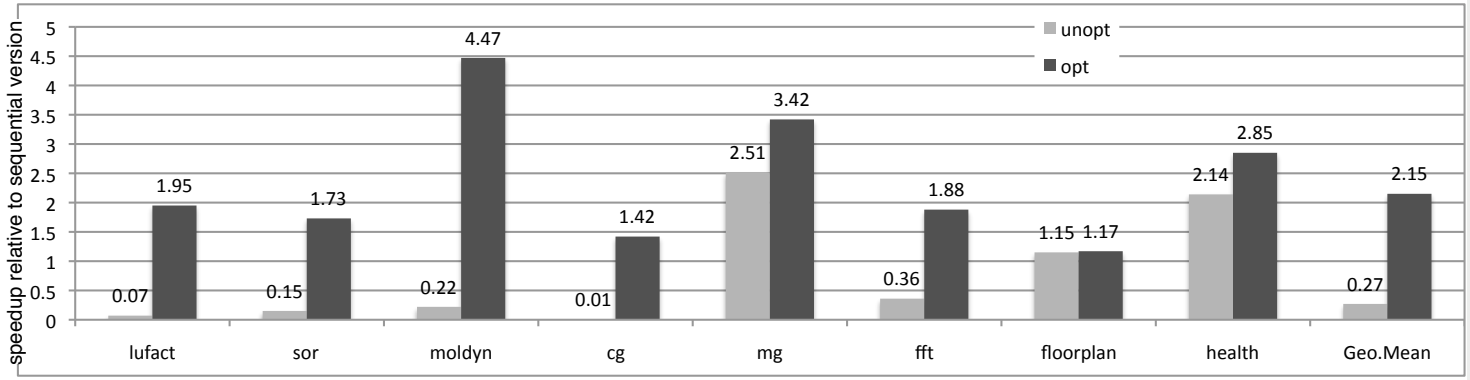
opt =
- after chunking



(a) T2



(b) Xeon



(c) Power7

RICE

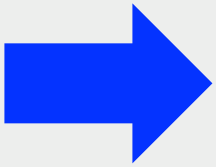
Three example optimizations for parallel tasks and parallel loops

1. Loop chunking
   - chunking fine-grain parallel loops into coarse-grained parallel tasks eliminates the significant overhead for task spawning and scheduling.
2. Forall coarsening
   - reduce task creation and termination overheads by increasing the scope of forall loops
     - *Simple forall-coarsening* increases the granularity of synchronization-free parallelism
     - *Forall-coarsening with synchronization* further increases the granularity of parallelism by adding synchronization operations (SPMDization)
3. Finish elimination
   - eliminate and/or reshape finish regions to reduce synchronization overhead and increase parallelism

Example of Illegal Forall Coarsening

```
1:  delta = epsilon+1; iters = 0;
2:  while (delta > epsilon) {
3:    forall (point[j] : [1:n]) {
4:      newA[j] = (oldA[j-1]+oldA[j+1])/2.0;
5:      diff[j] = Math.abs(newA[j]-oldA[j]);
6:    } // forall
      // sum and exchange
7:    delta = diff.sum(); iters++;
8:    temp=newA; newA=oldA; oldA=temp;
    } // while
```

(a)



```
1:  delta = epsilon+1; iters = 0;
2:  forall (point[j] : [1:n]) {
3:    while (delta > epsilon) {
4:      newA[j] = (oldA[j-1]+oldA[j+1])/2.0;
5:      diff[j] = Math.abs(newA[j]-oldA[j]);
        // sum and exchange
6:      delta = diff.sum(); iters++;
7:      temp=newA; newA=oldA; oldA=temp;
      } // while
    } // forall
```

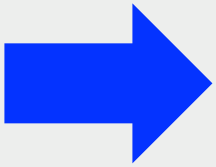
(b)

Naïve interchange of forall and while loops is illegal (no barrier leads to data races)

Example of Legal Forall Coarsening

```
1:  delta = epsilon+1; iters = 0;
2:  while (delta > epsilon) {
3:    forall (point[j] : [1:n]) {
4:      newA[j] = (oldA[j-1]+oldA[j+1])/2.0;
5:      diff[j] = Math.abs(newA[j]-oldA[j]);
6:    } // forall
      // sum and exchange
7:    delta = diff.sum(); iters++;
8:    temp=newA; newA=oldA; oldA=temp;
    } // while
```

(a)

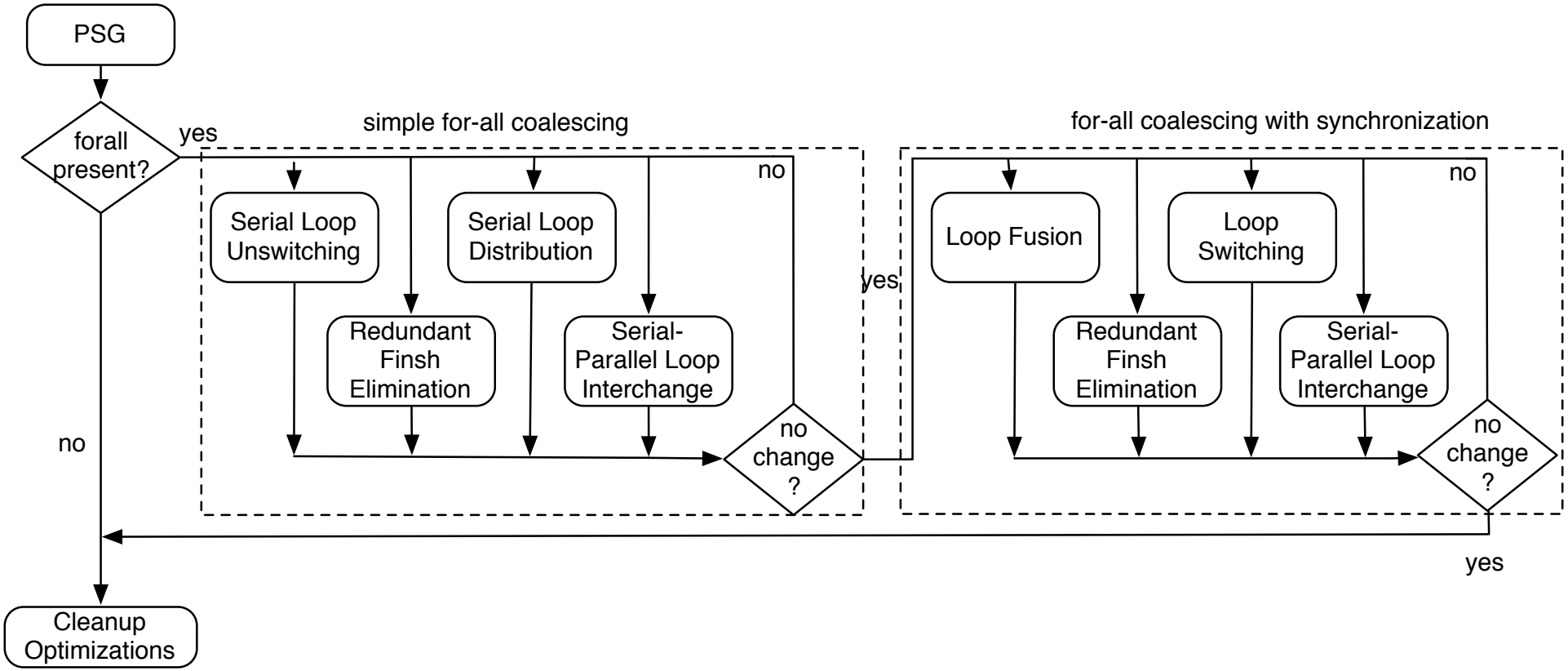


```
1:  delta = epsilon+1; iters = 0;
2:  forall (point[j] : [1:n]) {
3:    while (delta > epsilon) {
4:      newA[j] = (oldA[j-1]+oldA[j+1])/2.0;
5:      diff[j] = Math.abs(newA[j]-oldA[j]);
        // sum and exchange
6:      next single {
7:        delta = diff.sum(); iters++;
9:        temp=newA; newA=oldA; oldA=temp;
        } // next single
      } // while
    } // forall
```

(c)

Use of next barrier with single statement leads to a correct transformation (SPMDization)

Forall Coarsening Framework




```
for (int i=0;i<n;++i){
 S1;
 forall(point[j]:[1..m]){
  S2;
 }
 S3;
}
```
(a)

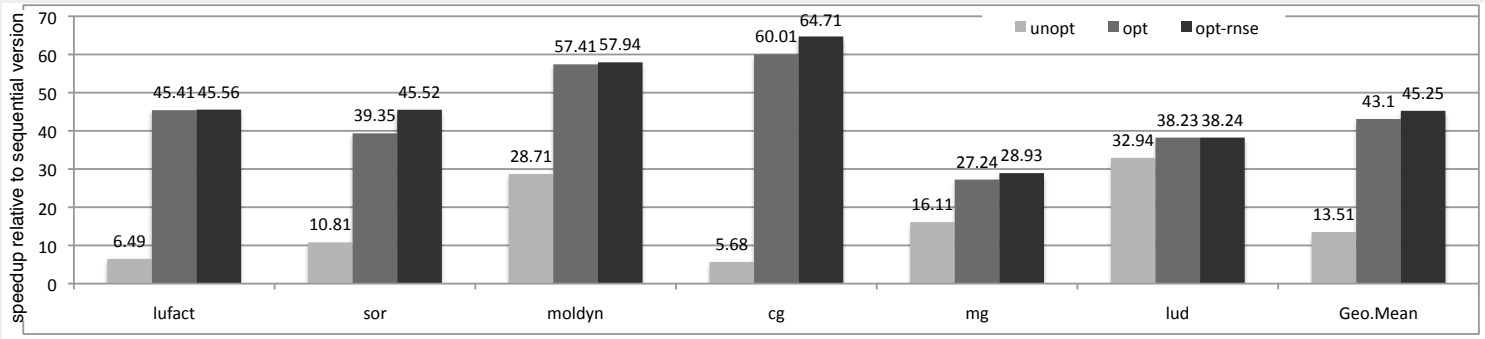
```
for (int i=0;i<n;++i){
 S1; }
forall(point[j]:[1..m]){
 for (int i=0;i<n;++i){
  S2; } }
for (int i=0;i<n;++i){
 S3; }
```
(b)

```
forall(point[j]:[1..m]){
 for (int i=0;i<n;++i){
  next S1;//next-single
  S2;
  next S3;//next-single
 }
}
```
(c)

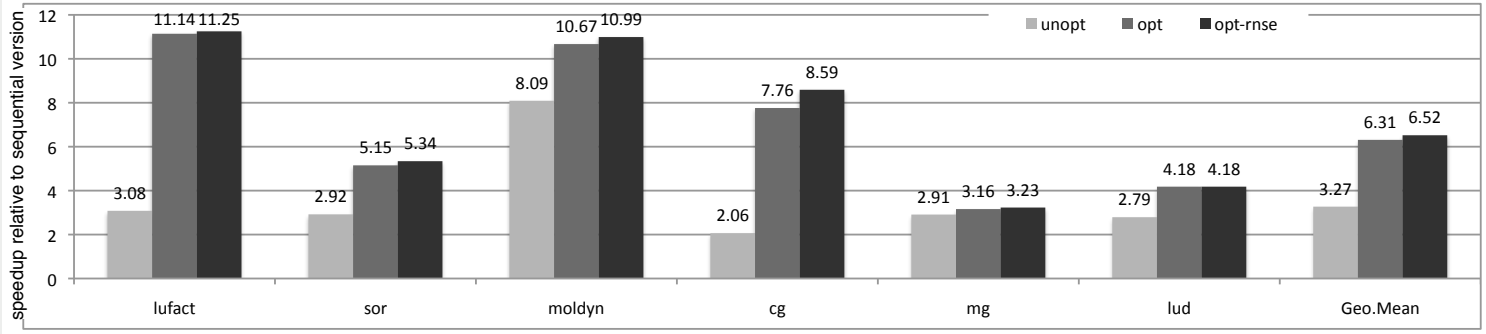
Performance Results for Forall Coarsening

unopt =
  chunking

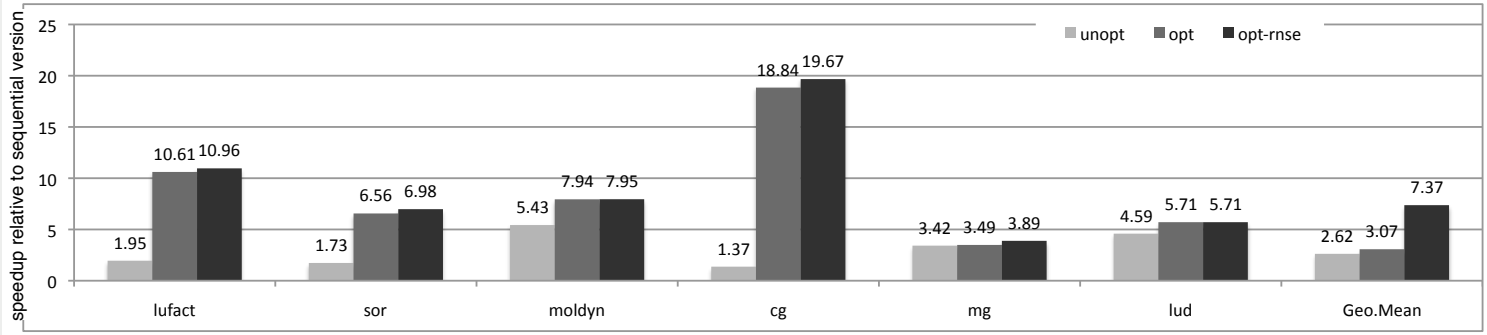
opt =
  chunking +
  coarsening



(a) T2



(b) Xeon



(c) Power7

opt-rnse = opt + Redundant Next/Single Elimination

Three example optimizations for parallel tasks and parallel loops

1. Loop chunking
   - chunking fine-grain parallel loops into coarse-grained parallel tasks eliminates the significant overhead for task spawning and scheduling.
2. Forall coarsening
   - reduce task creation and termination overheads by increasing the scope of forall loops
     - *Simple forall-coarsening* increases the granularity of synchronization-free parallelism
     - *Forall-coarsening with synchronization* further increases the granularity of parallelism by adding synchronization operations (SPMDization)
3. Finish elimination
   - eliminate and/or reshape finish regions to reduce synchronization overhead and increase parallelism

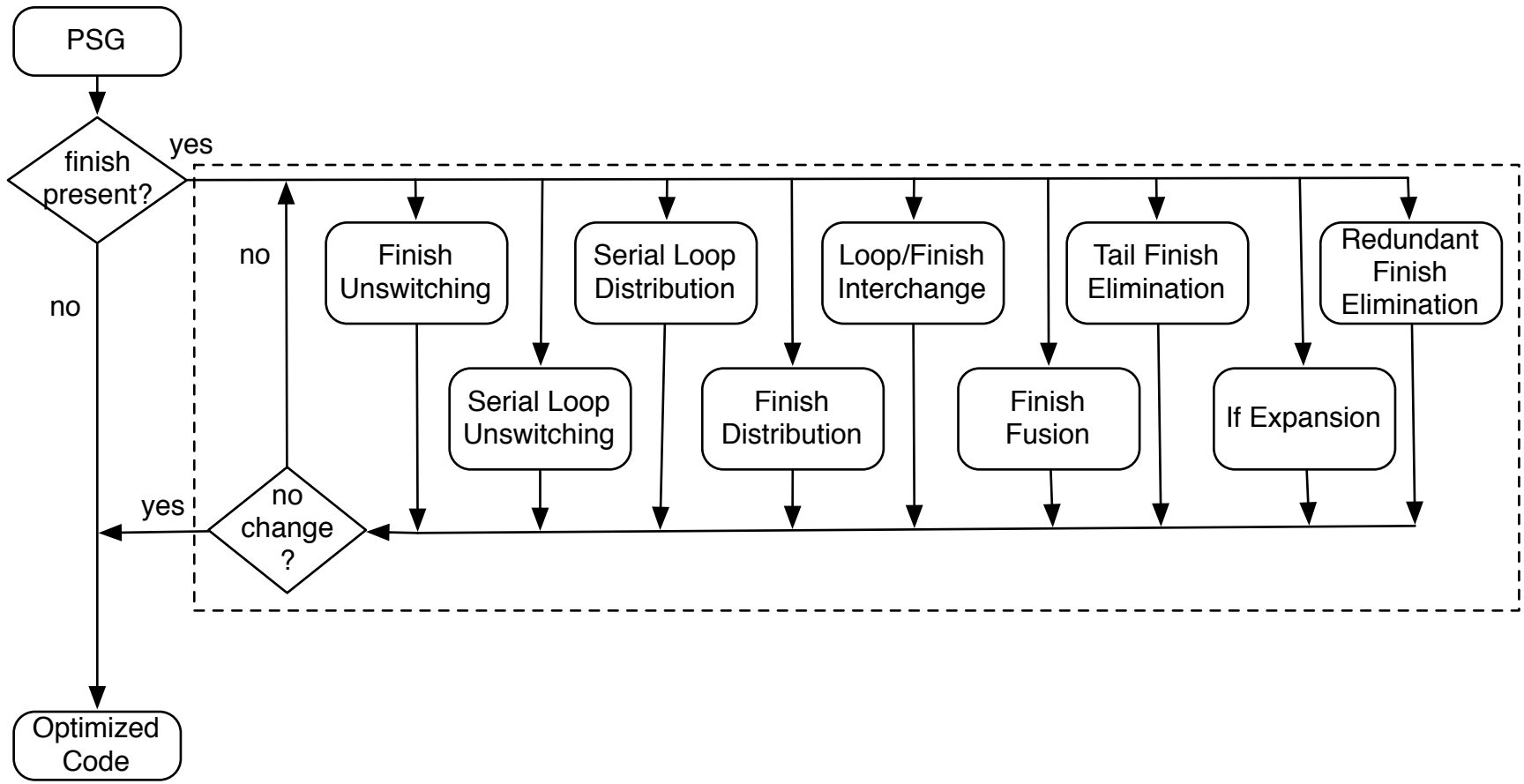
BOTS Health Benchmark with Recursive Asyncs

```
// Traverse village hierarchy
void sim_village_par(final Village village) {
    ...
1:  finish {
2:    final Iterator it=village.forward.iterator();
3:    while (it.hasNext()) {
4:      final Village v = (Village)it.next();
5:      // seq clause specifies threshold condition
        // async cannot have phased or await clauses
6:      async seq (sim_level - village.level >= bots_cutoff_value)
7:        sim_village_par(v);
      } // while
8:    ... ...;
9:  } // finish:
10: ... ...
```


Optimized Code after Finish Elimination

```
// Traverse village hierarchy
void sim_village_par(final Village village) {
    ...
1:  if (sim_level - village.level < bots_cutoff_value) {
2:    finish {
3:      final Iterator it=village.forward.iterator();
4:      while (it.hasNext()) {
5:        final Village v = (Village)it.next();
6:        async sim_village_par(v);
7:      } // while
8:      ... ...;
9:    } // finish
10: } else {
11:   final Iterator it=village.forward.iterator();
12:   while (it.hasNext()) {
13:     final Village v = (Village)it.next();
14:     sim_village_par(v);
15:   } // while
16:   ... ...;
17: }
18: ... ...
```


Finish Elimination Framework



Performance Results for Finish Elimination

unopt =

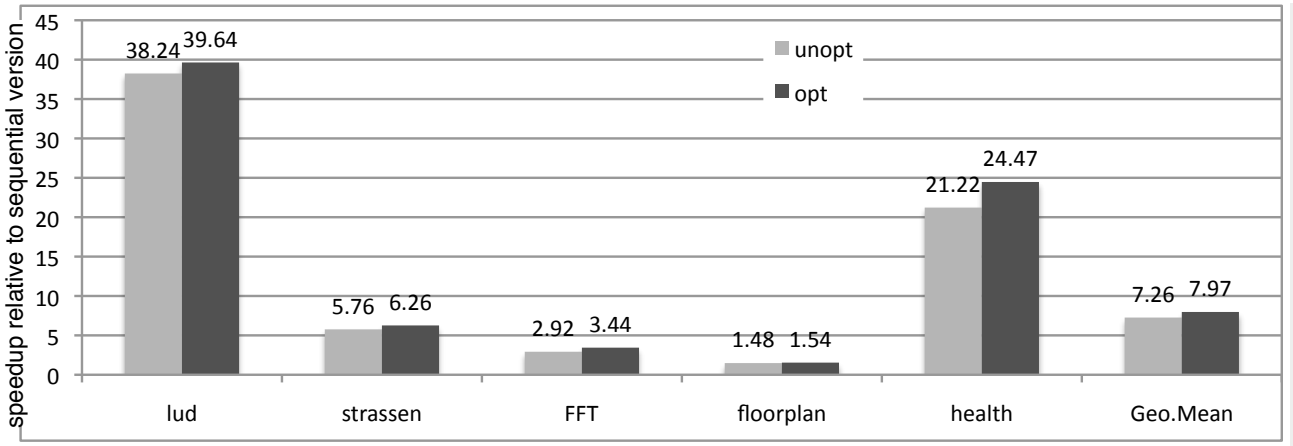
chunking +

coarsening

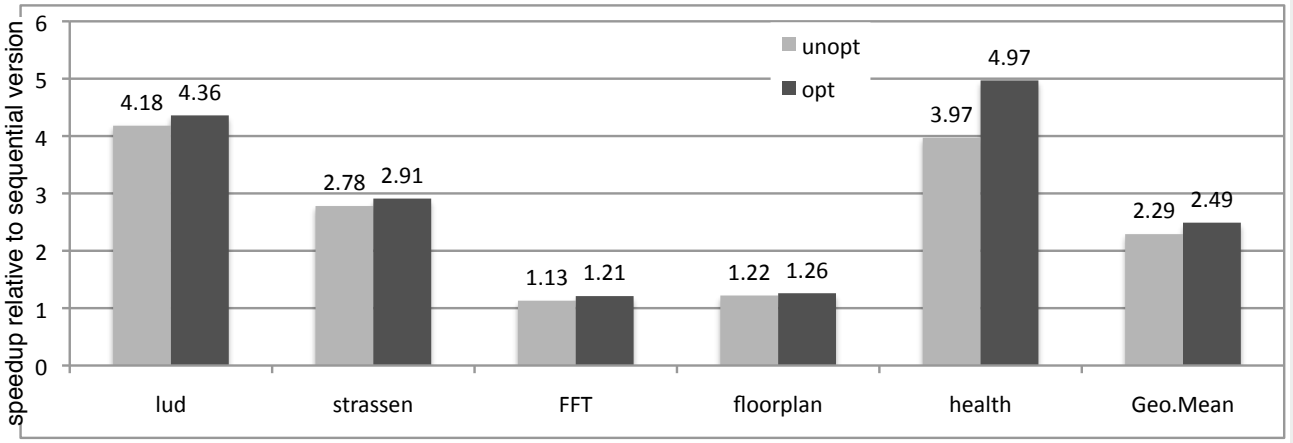
opt =

unopt +

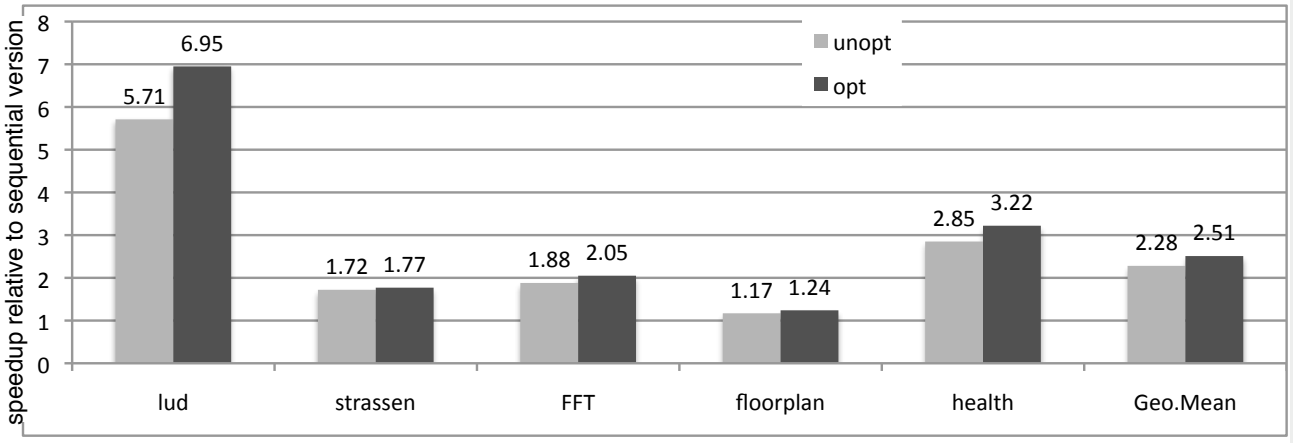
finish elim.



(a) T2



(b) Xeon



(c) Power7

RICE

Outline of Today's Lecture

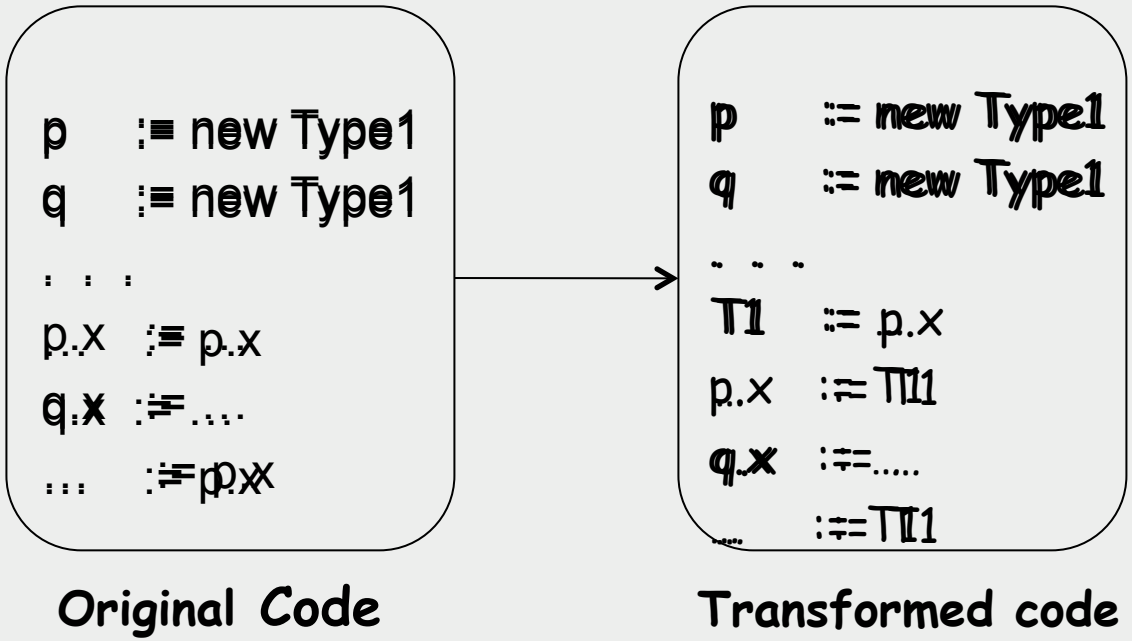
- HPIR Example: A Transformation Framework for Optimizing Task-Parallel Programs

- MPIR Example: Load Elimination

MPIR example: Load Elimination [BS09]

- Load Elimination is a compiler transformation that replaces a heap access by a read of a compiler-generated temporary
  - Temporary can be allocated on a faster/energy-efficient storage like register, scratchpads etc
- Best performed at medium PIR level
  - Flattened control flow simplifies data flow analysis (compared to HPIR)
  - Runtime-independent finish and async operators also simplifies analysis (compared to LPIR)

Load Elimination Example

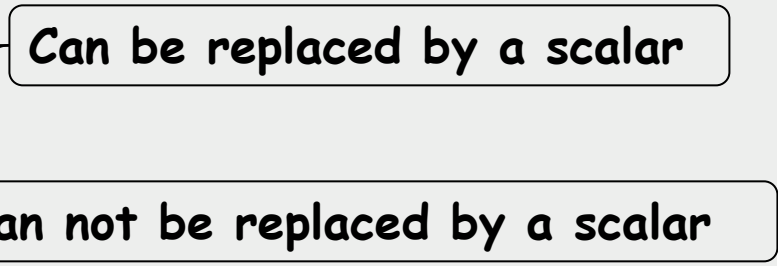


Example of Load Elimination Example in HJ

```
1: void main() {
2:   p.x = …
3:   s.w = …
4:   finish {
5:     async { //async_1
6:       p.x = …
7:       isolated { q.y = …; … = q.y }
8:       … = p.x
9:     } // async_1
10:    foo()
11:  } // finish
12:  … = p.x
13:  … = s.w
14: }

15: void foo() {
16:     async bar() //async_2
17:     isolated { q.y = … }
18:     … = s.w
19: }

20: void bar() {
21:     r.z = …
22:     .. = r.z
23: }
```



Side-Effect Analysis

- Effects of function calls
  - What variables may be modified as side effects of a function call?
- Extend Banning's formulation of side effects
  - MOD(*s*), REF(*s*): set of variables that may be modified/ referenced as a side effect of *s*
  - USE(s): set of variables that may be referenced as a side effect of *s* before being redefined
  - DEF(s): set of variables that must be modified as a side effect of *s*
  - GMOD(*p*), GREF(p): set of global variables and formal parameters *w* of *p* that are modified/referenced, either directly or indirectly as a result of function call of *p*

Side-Effects

- Async and normal method level side-effects
  - GMOD/GREF – Generalized modified/referenced side-effects
  - IMOD/IREF – Immediate modified/referenced side-effects
- Escaping async level side-effect
  - EMOD/EREF – Escaping modified/referenced side-effects
- Finish scope level side-effect
  - FMOD/FREF - modified/referenced side-effects for finish scope
- Atomic/Isolated level side-effect
  - AMOD/AREF – modified/referenced side-effects for isolated blocks

Side-Effects for Escaping Asyncs

- Async-Escaping Method Level Side-Effect (EMOD, EREF)
  - Sequential calls to methods that contain async constructs which are not wrapped in finish scopes
  - GMOD and GREF sets for async-escaping methods need to be propagated in the call chain to their immediate enclosing finish (*IEF*) scopes

```
1:  void foo () {
2:    async bar() // A
3:    … = p.x
4:    … = p.x
5:  }

6: void bar () {
7:   p.x = …
8: }

9:  void main () {
10:  p.x = …
11:  finish { // F
12:     foo ()
13:     … = p.x
14:  }
15:  … = p.x
16: foo ()
17: }
```

GMOD (bar) = {p.x}

GMOD (A) = {p.x}

GMOD (foo) = {}

EMOD (foo) = {p.x}

EMOD (main) = {p.x}

Side-Effects for Finish Scopes

- Finish Scope Level Side-Effect (FMOD, FREF)
  - Any async created within a finish scope scope must be completed before the statement after it is executed
  - FMOD and FREF side effects comprise of the heap accesses for the asyncs within the finish scope

```
1:  void foo () {
2:    async bar() // A
3:    … = p.x
4:    … = p.x
5:  }

6: void bar () {
7:   p.x = …
8: }

9:  void main () {
10:  p.x = …
11:  finish { // F
12:     foo ()
13:     … = p.x
14:  }
15:  … = p.x
16:  foo ()
17: }
```

GMOD (bar) = {p.x}
GMOD (A) = {p.x}
GMOD (foo) = {}
EMOD (foo) = {p.x}
EMOD (main) = {p.x}

FMOD (F) = {p.x}
GMOD (main) = {p.x}

Load Elimination and Memory Model

- Load elimination in the presence of parallel construct
  - Legality of transformation depends on memory model
  - All memory models have same semantics for data-race free programs
  - Compiler does not know if the input program is data-race free

Isolation Consistency Memory Model for HJ

- Isolation Consistency Memory Model
  - Builds on Location Consistency Memory Model [Gao & Sarkar '00]
  - State of a shared location is defined using a partially ordered multi-set (pomset) of write operations
  - A read operation sees a value that is
    - written by a most recent predecessor write
    - a write operation that is unrelated
  - Preserves control and data dependencies within a thread
  - Weaker than sequential consistency
    - Intended for application code rather than systems code

IC Memory Model Examples

```
1: A a = new A ()
2: a.f = …
3: async { … }
4: … = a.f
```

Case 1

```
1: final A a = new A ()
2: a.f = …
3: async { while(…) a.f = F(a.f) }
4: … = a.f
```

Case 2

```
1: final A a = new A ()
2: a.f = …
3: finish async { a.f = … }
4: … = a.f
```

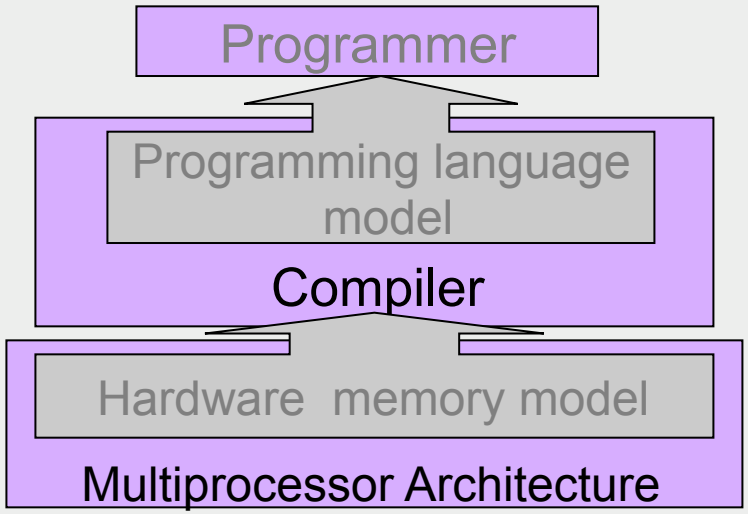
Case 3

```
1: final A a = new A ()
2: a.f = …
3: async { isolated if (…) a.x++ }
4: … = a.f
```

Case 4

RICE

The Compiler's task



- Compiler must enforce programming language memory model
  - Hardware and software model may differ
  - If language model is weaker than hardware model, then compiler may have opportunities for code optimization
  - If hardware model is weaker than language model, then compiler may need to add synchronization operations (*fences*) to support language semantics

Summary of MPIR-level Load Elimination Algorithm

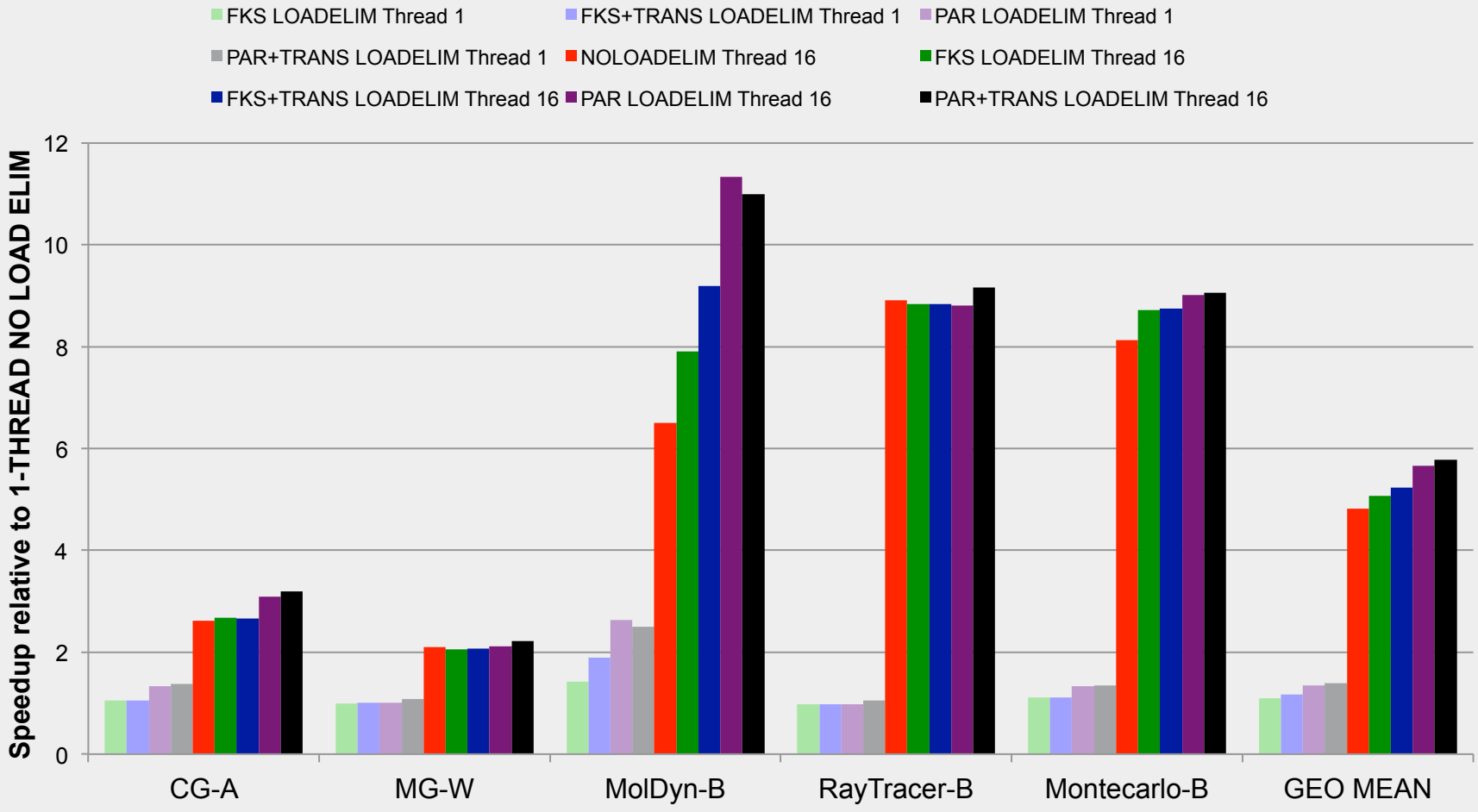
- Compute side-effects for each function call, finish scope and global isolated level using side-effect analysis described before
- Append pseudo-defs and pseudo-uses to fields based on side-effects and isolation consistency memory model
- Create heap operands for the pseudo-defs and pseudo-uses
- Construct extended array-ssa form for the heap operands
- Perform global value numbering to compute Definitely-Same (*DS*) and Definitely-Different (*DD*) relations
- Perform data flow analysis to propagate uses to defs
- Eliminate loads if the value number is available

Reduction in Dynamic Field Accesses

Benchmarks	# getfield original	#getfield after FKS Load elim.	#getfield after FKS +TRANS Load elim.	#getfield after PAR Load elim.	#getfield after PAR +TRANS Load elim.	Impr. relative to Original (%)	Impr. Relative to FKS	Impr. Relative to FKS +TRANS
CG-S	3.89E09	3.10E09	3.03E09	2.34E09	3.92E05	99.99%	99.99%	99.99%
MG-W	1.41E04	1.15E04	1.13E04	7.96E03	6.71E03	52.55%	41.72%	40.58%
MolDyn-B	1.19E10	7.91E09	5.82E09	4.91E09	3.11E09	73.89%	60.62%	46.49%
RayTracer-B	3.08E10	2.02E10	2.02E10	1.67E10	1.38E10	55.25%	31.93%	31.82%
Montecarlo-B	1.75E09	1.54E09	1.48E09	5.84E08	9.19E08	47.38%	40.48%	37.95%
specJBB	1.19E09	1.02E09	8.95E08	6.65E08	5.78E08	51.56%	43.19%	35.43%

Decrease in dynamic counts of getfield operations of upto ~ 1000x

Speedup on 4 Quadcore Intel Xeon



Runtime improvement: up to 1.76× on 1 core, and 1.39× on 16 cores

Conclusions

- Theoretical foundations of parallel programs have historically been defined using unstructured parallel programming constructs such as threads and locks
- This talk presented early experiences in the Habanero project on identifying structured parallelism primitives that provide a foundation for programmability, compilation and runtime
- The benefits of structured parallelism were illustrated using examples from program analysis and transformation

A Call to Arms --- creating a general compiler back-end framework for Optimizing Parallel Programs

- Extend LLVM IR with support for PGAS languages with explicit task parallelism e.g.,
  - Chapel, X10
  - Habanero-C with distributed futures
  - Habanero-UPC (Rice-LBL collaboration)
- Other source languages
  - Output from a DSL?
  - UPC (current standard doesn't have task parallelism)
  - CAF (current standard doesn't have task parallelism)
  - OpenMP (separate effort under way for LLVM extensions)
- [illegible]

Send email to Vivek Sarkar (vsarkar@rice.edu) if you are interested in a PhD, postdoc or research scientist position in the Habanero project, or in collaborating with us!