Outline of Tutorial

1) Overview of task-parallel languages
   - Cilk, OpenMP 3.0, Chapel, X10, Habanero-Java (HJ)
2) Optimizations of HJ programs at the High-level Parallel Intermediate Representation (HPIR) level
   - May-Happen-in-Parallel (MHP) analysis
   - Forall coarsening
   - Forall chunking
   - Finish elimination
3) Optimizations of HJ programs at the Middle-level (MPIR) and Low-level (LPIR) Parallel Intermediate Representations
   - Load elimination
   - Optimizations for work-stealing runtime schedulers
4) Communication optimizations of X10 programs on distributed-memory parallel machines
Acknowledgments

- PLDI 2008 tutorial on “Analysis and Optimization of Parallel Programs”, Sam Midkiff and Vivek Sarkar
- PLDI 2007 tutorial on X10, Vijay Saraswat, Vivek Sarkar, Nathaniel Nystrom
- Additional X10 slides from Vijay Saraswat, David Grove, et al
- Chapel slides from Brad Chamberlain
- Research collaborators on optimization of HJ programs
  - Rice: Raj Barik, Jun Shirako, Jisheng Zhao
  - IBM: Krishna Nandivada

Parallel Programming Models: A Brief Taxonomy

- 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)
Introducing Cilk [BJK+95]

Cilk int fib (int n) {
  if (n<2) return (n);
  else {
    int x,y;
    x = spawn fib(n-1);
    y = spawn fib(n-2);
    sync;
    return (x+y);
  }
}

Identifies a function as a Cilk procedure (task) capable of being spawned in parallel.

The named child Cilk procedure can execute in parallel with the parent caller (fork).

Control cannot pass this point until all spawned children have returned (join).

Terminology

- **Parallel control** = spawn, sync, return from spawned function
- **Step** = maximal sequence of instructions not containing parallel control (referred to as a “thread” in Cilk papers)

Cilk int fib(n) {
  if (n < 2) return n;
  else {
    int n1, n2;
    n1 = spawn fib(n-1);
    n2 = spawn fib(n-2);
    sync;
    return (n1 + n2);
  }
}

- **Step A**: if statement up to first spawn
- **Step B**: computation of n-2 before 2nd spawn
- **Step C**: n1+ n2 before the return

“continue” edges between steps in same dynamic task:
Computation Graph

- The computation graph $G = (V, E)$ represents a dynamic execution of a Cilk program.
- $G$ is a directed acyclic graph (dag)
- Each vertex $v$ in $V$ represents a (Cilk) step: a maximal sequence of instructions not containing parallel control (spawn, sync, return).
- Every edge $e$ in $E$ is either a spawn edge, a return edge, or a continue edge.

Task-Scheduling Paradigms [FLR98, GBRS09, GZCS10]

- Work-Sharing (e.g., Java Concurrency…)
  - One double-ended queue (deque) shared by all workers
  - Busy worker inserts new task in shared deque
  - Idle worker pulls new task from shared deque
  - Accesses to the shared deque need to be synchronized: scalability bottleneck

- Work-Stealing (e.g., Cilk, Habanero-Java)
  - Each worker has its own deque
  - Additional work is pushed onto worker’s deque
  - Idle worker steals the tasks from busy workers with minimum impact on busy workers.
  - Better scalability than work-sharing
Introducing OpenMP

- Popular standard for writing shared-memory parallel programs in C, C++, Fortran
  - Original motivation: efficient support for loop parallelism
- OpenMP consists of
  - Compiler directives (pragmas)
  - Runtime routines
  - Environment variables
- Specification maintained by the OpenMP Architecture Review Board (http://www.openmp.org)
  - Latest specification: Version 3.0 (May 2008)
  - Previous specification: Version 2.5 (May 2005)

The OpenMP Execution Model a.k.a. Single-Program Multiple-Data (SPMD)
OpenMP 3.0 task Construct
(similar to Cilk’s spawn)

```c
#pragma omp task [clause[[],clause] ...]

structured-block

where clause can be one of:

- if (expression)
- untied
- shared (list)
- private (list)
- firstprivate (list)
- default( shared | none )
```

OpenMP tasks are “tied” to worker threads by default

When/where are tasks complete?

- At taskwait operations (like Cilk’s sync)
  - applies only to child tasks generated in the current task,
    not to “descendants”
  - #pragma omp taskwait

- At thread barriers, explicit or implicit
  - applies to all tasks generated in the current parallel region up to the barrier
  - matches user expectation
Example #1: Parallel execution of a pointer-chasing loop using tasks

```c
#pragma omp parallel
{
    #pragma omp single private(p)
    {
        p = listhead;
        while (p) {
            #pragma omp task
            process (p)
            p = p->next;
        }
    }
}
```

Spawn call to `process(p)`

Implicit taskwait

Example #2: Parallel execution of multiple pointer-chasing loops using tasks (nested parallelism)

```c
#pragma omp parallel
{
    #pragma omp for private(p,i)
    for (int i = 0; i < numlists; i++) {
        p = listheads[i];
        while (p) {
            #pragma omp task
            process (p)
            p = next(p);
        }
    }
}
```
Example #3: Postorder Tree Traversal

```c
void postorder(node *p) {
    if (p->left)
        #pragma omp task
        postorder(p->left);
    if (p->right)
        #pragma omp task
        postorder(p->right);
    #pragma omp taskwait // wait for child tasks
    process(p->data);
}
```

- Parent task suspends until children tasks complete

Chapel: Task Parallelism Terminology

**Task**: a unit of parallel work in a Chapel program
- all Chapel parallelism is implemented using tasks
- `main()` is the only task when execution begins

**Thread**: a system-level concept that executes tasks
- not exposed in the language
- occasionally exposed in the implementation

See [http://chapel.cray.com](http://chapel.cray.com) for details
Block-Structured Task Creation: Cobegin

- Syntax
  
  `cobegin-stmt:
  cobegin { stmt-list }

- Semantics
  - Creates a task for each statement in `stmt-list`
  - Parent task waits for `stmt-list` tasks to complete

- Example

```
cobegin {
  consumer(1);
  consumer(2);
  producer();
}
// wait here for both consumers and producer to returned
```

Loop-Structured Task Invocation: Coforall

- Syntax
  
  `coforall-loop:
  coforall index-expr in iterable-expr { stmt-list }

- Semantics
  - Create a task for each iteration in `iteratable-expr`
  - Parent task waits for all iteration tasks to complete

- Example

```
begin producer();
coforall i in 1..numConsumers {
  consumer(i);
}
// wait here for all consumers to returned
```
X10 Programming Model

- X10 is an object-oriented Partitioned Global Address Space (PGAS) language with support for distributed asynchronous dynamic parallelism
  - Lightweight dynamic activity creation and termination
    - Fine grained concurrency: async S;
    - Place-shifting operations: at (place) async S
    - Synchronization: finish S;
  - Locality control --- task and data distributions
    - Places
    - Distributed arrays (Array)
      - Point, Region, Dist
      - Maps every point in a region using a distribution, e.g. block, cyclic
    - Mutual exclusion
      - atomic S; per-place based
- See http://x10-lang.org for details

X10 Constructs

<table>
<thead>
<tr>
<th>Fine grained concurrency</th>
<th>Atomicity</th>
<th>Global data-structures</th>
</tr>
</thead>
<tbody>
<tr>
<td>• async S</td>
<td>• atomic S</td>
<td>• points, regions, distributions, arrays</td>
</tr>
<tr>
<td>• when (c) S</td>
<td>• when (c) S</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Place-shifting operations</th>
<th>Ordering</th>
</tr>
</thead>
<tbody>
<tr>
<td>• at (P) S</td>
<td>• finish S</td>
</tr>
<tr>
<td></td>
<td>• clocked, next</td>
</tr>
</tbody>
</table>

Two basic ideas: Places and Asynchrony
Habanero-Java (HJ) Language

- HJ is a new language developed in the Rice Habanero Multicore Software research project (http://habanero.rice.edu/hj)
  - Download from http://habanero.rice.edu/hj-download
  - Derived from IBM’s Java-based X10 v1.5 implementation in 2007
  - HJ is an extension of Java 1.4
    - Java 5 & 6 language features (generics, metadata, etc.) are currently not supported by the HJ front-end
    - However, Java 5 & 6 libraries and classes can be called from HJ programs
      - Just don’t call a method that performs a blocking operation because that will mess up the HJ scheduler!

- Four classes of parallel programming primitives in HJ:
  1. Dynamic task creation & termination: forall, async, finish, get
  2. Mutual exclusion and isolation: isolated
  3. Collective and point-to-point synchronization: phaser, next
  4. Locality control --- task and data distributions: places, here

Async and Finish Statements for Task Creation and Termination

async S
- Creates a new child task that executes statement S
- Parent task immediately continues to statement following the async

```java
//Task T₀ (Parent)
finish {
    async
    STMT1; //T₁ (Child)
    //Continuation
    STMT2; //T₀
} //Continuation //End finish
STMT3; //T₀
```

finish S
- Execute S, but wait until all (transitively) spawned asyncs in S’s scope have terminated.
- Implicit finish between start and end of main program

```
T₁ T₀
---------------------
async
STMT1 STMT2

terminate

wait

STMT3
```
Comparing Async-Finish with Cilk’s Spawn-Sync

- Async-Finish permits sequential and parallel calls to the same function
- Async-Finish permits an async to “escape” a function call
- Async-Finish computations are more general than Cilk’s fully strict computations

```
l1 S0;
l2 finish { //startFinish
l3 async {
l4 S1;
l5 async {
l6 S2;}
l7 S3;}
l8 } //stopFinish
l9 S4;
```

Figure 3.1: HJ computation dag

3.2.2 Computation Classes

The computation dag model is very general. In practice, we are interested in a particular subset of computation dags that can be generated by a task parallel language, and those computation dags that can be scheduled efficiently.

One class of computations is called strict computations. Strict computations are those sync edges can only go from an instruction to one of its ancestors in the task spawn tree. Unlike functional programming languages, in Habanero-Java and Cilk tasks cannot be passed as an argument. So a task can only force the completion of a task that it spawns recursively. As a result, all computations generated by HJ and Cilk are strict computations. In full X10, however, it is possible to produce non-strict computations by passing futures as a parameter to its descendent tasks.

Fully-strict computation is a subset of strict computation. In fully-strict computations, the sync edge of a task can only go to its parent. All computations generated by Fully-strict computation

HJ isolated statement

isolated <body>

- Two tasks executing isolated statements with interfering accesses must perform the isolated statement in mutual exclusion
  - Two instances of isolated statements, \langle stmt1 \rangle and \langle stmt2 \rangle, are said to interfere with each other if both access a shared location, such that at least one of the accesses is a write.
  - Weak isolation guarantee: no mutual exclusion applies to non-isolated statements i.e., to (isolated, non-isolated) and (non-isolated, non-isolated) pairs of statement instances
- Isolated statements may be nested (redundant)
- Isolated statements must not contain any other parallel statement: async, finish, get, forall
- In case of exception, all updates performed by <body> before throwing the exception will be observable after exiting <body>
Example of Escaping Asyncs: Parallel Depth-First Search Spanning Tree (PDFS) [GZCS10]

class V {
    V[] neighbors;
    V parent;
    ...
    boolean tryLabeling(V n) {
        isolated if (parent == null) parent = n;
        return parent == n;
    }
}

void compute() {
    for (int i = 0; i < neighbors.length; i++) {
        V child = neighbors[i];
        if (child.tryLabeling(this))
            async e.compute(); // escaping async
    }
}

void DFS() {
    parent = this;
    finish compute();
}

} // class V

Barrier Synchronization: HJ’s “next” statement in forall construct

rank.count = 0; // rank object contains an int field, count
forall (point[i] : [0:m-1]) {
    int r;
    isolated {r = rank.count++;}
    System.out.println("Hello from task ranked "+r);
    next; // Acts as barrier between phases 0 and 1
    System.out.println("Goodbye from task ranked "+r);
}

- 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 (will be covered in following lectures)
### HJ Compilation and Execution Environment

- **HJ source program** — must contain a class named `Foo` with a `public static void main(String[] args)` method.
- **hjc Foo.hj**
  - **HJ compiler**
  - **Foo.class**
- **hj -places m:n Foo**
  - **HJ Runtime Environment = JRE + HJ libraries + HJ Multithreaded Runtime**
  - **HJ Program Output**

**Caveat:** this is a research prototype with known limitations.

### Comparison of Task-Parallel Programming Models along Selected Dimensions

<table>
<thead>
<tr>
<th></th>
<th>Dynamic Parallelism</th>
<th>Locality Control</th>
<th>Mutual Exclusion</th>
<th>Collective &amp; Point-to-point Synchronization</th>
<th>Data Parallelism</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cilk</td>
<td>Spawn, sync</td>
<td>None</td>
<td>Locks</td>
<td>None (null)</td>
<td>None</td>
</tr>
<tr>
<td>Java Concurrency</td>
<td>Executors, Task Queues</td>
<td>None</td>
<td>Locks, monitors, atomic classes</td>
<td>Synchronizers</td>
<td>Concurrent collections</td>
</tr>
<tr>
<td>Intel TBB</td>
<td>Generic alg, tasks</td>
<td>None</td>
<td>Locks, atomic classes</td>
<td>None</td>
<td>Concurrent containers</td>
</tr>
<tr>
<td>.Net Parallel Extensions</td>
<td>Generic alg, tasks</td>
<td>None</td>
<td>Locks, monitors</td>
<td>Futures</td>
<td>PLINQ</td>
</tr>
<tr>
<td>OpenMP</td>
<td>SPMD (v2.5), Tasks (v3.0)</td>
<td>None</td>
<td>Locks, critical, atomic</td>
<td>Barriers</td>
<td>None</td>
</tr>
<tr>
<td>CUDA</td>
<td>None</td>
<td>Device, grid, block, threads</td>
<td>Device-memory locks</td>
<td>Intra-block barriers</td>
<td>SIMT</td>
</tr>
<tr>
<td>X10</td>
<td>Async, finish</td>
<td>Places</td>
<td>Place-local atomic</td>
<td>Clocks, futures</td>
<td>First-class arrays, regions, distributions</td>
</tr>
<tr>
<td>Habanero-Java</td>
<td>Async, finish</td>
<td>Hierarchical Places</td>
<td>Multi-place isolated</td>
<td>Phasors, futures, data-driven futures</td>
<td>None</td>
</tr>
</tbody>
</table>
Classification of Task-Parallel Programs

- Legend
  - DET = Deterministic
  - DRF = Data-race-free
  - DLF = Deadlock-free
  - SER = Serializable

- Subsets of task-parallel constructs can be used to guarantee membership in certain classes e.g.,
  - If an HJ program is data-race-free and only uses the async and finish constructs, then it is guaranteed to belong to the DLF-DRF-DET-SER class

Switching from the Programmer’s Viewpoint to the Compiler’s Viewpoint

- Consider a basic compiler analysis problem for parallel programs --- May-Happen-in-Parallel (MHP) analysis
  - Given two statements S1 and S2 in a parallel program, determine if it’s possible for an instance of S1 to execute in parallel with an instance of S2

- Things to think about
  - What intermediate representation would you prefer to use for MHP analysis? An Abstract Syntax Tree (AST)? A flat three-address intermediate representation? Something in between?
  - What algorithmic approach would you use for MHP analysis?
  - How precise can the MHP analysis be? Can you identify example programs where the compile asserts that MHP(S1, S2) = true, but it isn’t possible for instances of statement S1 and S2 to execute in parallel with each other?
  - What is the compile-time complexity of the MHP Analysis algorithms that you can think of?
Outline of Tutorial

1) Overview of task-parallel languages
   - Cilk, OpenMP 3.0, Chapel, X10, Habanero-Java (HJ)
2) Optimizations of HJ programs at the High-level Parallel Intermediate Representation (HPIR) level
   - May-Happen-in-Parallel (MHP) analysis
   - Forall coarsening
   - Forall chunking
   - Finish elimination
3) Optimizations of HJ programs at the Middle-level (MPIR) and Low-level (LPIR) Parallel Intermediate Representations
   - Load elimination
   - Optimizations for work-stealing runtime schedulers
4) Communication optimizations of X10 programs on distributed-memory parallel machines

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
HPIR example: HJ Program Structure Tree (PST)

- The PST for an HJ procedure is a rooted tree with six types of nodes
  - **Root node** --- represents entire procedure
  - **Async node** --- represents an async statement
    - Async node may be annotated with destination place expression
  - **Finish node** --- represents a finish statement
  - **Isolated node** --- represents an isolated statement
  - **Loop node** --- represents a sequential loop statement
    - A parallel loop is modeled as a sequential loop with an async body
  - **Other** statement --- represents a leaf node in the PST
- Parent relation in PST is determined by program structure
  - PST.parent(N) is the node that represents the closest enclosing async/finish/atomic/loop statement (or root node if none)

---

Example of MHP Analysis for Java [NAC99]
(SplitRendererNested example in Figure 3)

<table>
<thead>
<tr>
<th>Main thread:</th>
<th>ExternalHelper1 thread:</th>
<th>InternalHelper1_1:</th>
</tr>
</thead>
</table>
| S1: ExternalHelper1.start(); | S5: ... | S11: ...
| S2: ... | S6: InternalHelper1_1.start(); | S11: ...
| S3: ExternalHelper1.join(); | S7: InternalHelper1_2.start(); | InternalHelper1_2: ...
| S4: ... | S8: InternalHelper1_1.join(); | S12: ...
| S9: InternalHelper1_2.join(); | S10: ... |

// Algorithm in [18] computes
// MHP(S4,S11) = true,
// and MHP(S4,S12) = true

// Conservative (imprecise) solution computed for MHP(S10,S11) and MHP(S10,S12)

Precise solution computed for MHP(S10,S11) and MHP(S10,S12)
May-Happen-in-Parallel (MHP) analysis
[Tay83a, Tay83b, CS88, DS91, MR93, NAC99, ABSS07]

- MHP(S1,S2) = true if statements S1 and S2 may execute in parallel
  - Foundation for static analysis and debugging tools for parallel programs
- Past algorithms for MHP analysis were slow …
  - NP-hard problem for Ada’s rendezvous primitive [Tay83a, Tay83b]
  - O((pN)^3) algorithm for Java [NAC99]
    - N: number of interprocedural control flow graph nodes
    - p: number of runtime threads
- … and imprecise
  - Limited by inter-procedural alias analysis capabilities, even for simple concurrency patterns in Java
- Observation: efficiency and precision of MHP analysis can be improved if applied to a high-level PIR

HPIR-based MHP Analysis for SplitRendererNested in HJ

S0: finish {
  S1: async {
    finish {
      S5: ...
      S6: async S11
      S7: async S12
      ...
    }
    S8: ...
    S9: ...
    S10: ...
  }
}

// MHP(S10,S11) = MHP(S10,S12) = false
}

S2: ...
S3: ...
S4: ...

Program Structure Tree

Execution-time complexity: O(H) for demand driven, O(N^H) for all pairs
H = height of PST, N = number of PST nodes
MHP inside Loops

A := ... // 3-d Array
for ( i = 1 ; i < n ; i++ )
  finish {
    for ( j = 0 ; j < n ; j++ )
      for ( k = 0 ; k < n ; k++ )
        async {
          /* S1 */
          ... = f(A[i-1,j,k])
          /* S2 */
          A[i,j,k] = ...
        } // async
  } // finish

• We need a way to distinguish MHP information for individual loop iterations
• Use condition vector sets (CS) -- akin to direction and distance vectors
• MHP(S1,S2) = true with CS = {(=, *, *)}

Another Example of MHP analysis on the HPIR PST

for ( i = 1 ; i <= n ; i++ )
  finish
  for ( j = 1 ; j <= n ; j++ )
    for ( k = 1 ; k <= n ; k++ )
      async {
        if (…) {
          /* S1 */
          ... = f(A[i,j,k])
        }
        /* S2 */
        A[i,j,k] = ...
      } // async

MHP(S1 ,S2 ) = false with condition vector set, CS = {
  (=, =, =),
  (≠, *, *)
}
i.e., MHP(S1 ,S2 ) = false if instances of S1 and S2 belong to the same i-j-k iteration, or if they come from iterations with distinct values of i
Coarsening and Chunking of Parallel Loops

- 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.
- Forall-chunking: extract useful parallelism by grouping together chunks of parallel iterations into separate tasks
- Extensions of past work on SPMDization and Loop Chunking
  - Input program may contain task-parallel operations within a forall loop
  - Input program may need to obey precise exception semantics

Transformation Rules used in HPIR Optimizations

<table>
<thead>
<tr>
<th>Rule</th>
<th>Transformation</th>
</tr>
</thead>
</table>
| 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; }  
⇒  
forall (point p : R1) S1;  
forall (point p : R1) S2; |
| 3. Loop/Finish Interchange: | for (S1; cond; S2)  
finish S3;  
// Say E_c = set of e-async in S3  
// ⇒ ∀e ∈ E_c: cond has dependence on e  
// ∀e ∈ E_c: body of e has loop  
// carried dependence on S2 or S3  
⇒  
finish  
for (S1; cond; S2)  
S3; |
| 4. Serial-parallel loop interchange: | for (i: [1..n])  
forall (point p : R1) S;  
// Different 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 |
### Transformation Rules used in HPIR Optimizations (contd)

#### 6. Loop Unpeeling:

<table>
<thead>
<tr>
<th>Rule</th>
<th>Condition</th>
<th>Transformation</th>
</tr>
</thead>
<tbody>
<tr>
<td>for all (point p: R) S1; S2; // no break/continue in S2.</td>
<td>for all (point p: R) S1; S2;</td>
<td></td>
</tr>
</tbody>
</table>

#### 7. Loop Fusion:

<table>
<thead>
<tr>
<th>Rule</th>
<th>Condition</th>
<th>Transformation</th>
</tr>
</thead>
<tbody>
<tr>
<td>for all (point p: R1) S1; for all (point p: R2) S2;</td>
<td>if (R1.contains (p) S1; next; if (R1.contains (p) S2; }</td>
<td></td>
</tr>
</tbody>
</table>

#### 8. Loop Switching:

<table>
<thead>
<tr>
<th>Rule</th>
<th>Condition</th>
<th>Transformation</th>
</tr>
</thead>
<tbody>
<tr>
<td>if (c) for all (point p: R) S;</td>
<td>final boolean v = c; for all (point p: R) S; if (v) S;</td>
<td></td>
</tr>
</tbody>
</table>

#### 9. Parallel Loop Unswitching:

<table>
<thead>
<tr>
<th>Rule</th>
<th>Condition</th>
<th>Transformation</th>
</tr>
</thead>
<tbody>
<tr>
<td>for all (point p: R1) if (e) // e is independent of p S</td>
<td>if (e) for all (point p: R1) S</td>
<td></td>
</tr>
</tbody>
</table>

#### 10. Serial Loop Unswitching:

<table>
<thead>
<tr>
<th>Rule</th>
<th>Condition</th>
<th>Transformation</th>
</tr>
</thead>
<tbody>
<tr>
<td>for(S2;cond1;S3){ if (cond2) S4; else S5; } // cond2 has no dependence // on S2, S3, S4 and S5, // cond2 has no side effects</td>
<td>if (cond2) { for(S2;cond1;S3) S4; } else { for(S2;cond1;S3) S5; }</td>
<td></td>
</tr>
</tbody>
</table>

### 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:

1. (Sequential order) If $I_A$ and $I_B$ belong to the same task, and $I_B$ is sequentially control or data dependent on $I_A$, $I_A$ and $I_B$ have control/data dependency and $I_A$ is executed before $I_B$, then $HB(I_A, I_B) = true$.
2. (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 $HB(I_A, I_B) = true$.
3. (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 $HB(I_A, I_B) = true$.
4. (Isolated) All instances of interfering isolated blocks in a dynamic execution of an HP program can be assumed to be serialized in some total order. If $I_A$ is the last statement in an isolated block instance and $I_B$ is the first statement of the next isolated block instance in the total order, then $HB(I_A, I_B) = true$.
5. (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 \( \text{HBD}(A, B) = \text{true} \) 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. \( \text{HB}(I_A, I_B) = \text{true} \),
  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 \( \text{HB}(I_A, I_C) = \text{true} \) and \( \text{HB}(I_C, I_B) = \text{true} \).

- 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.

Example of Illegal Forall Coarsening

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

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

Overview of Coalescing

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

for (int i = 0; i < n; ++i)
    S1;
for (int i = 0; i < n; ++i)
    S2;
    next S1;
    S2;
    next S3;
for (int i = 0; i < n; ++i)
    S3;
Overview of Forall Coarsening Framework

- Loop Unswitching
- Finish Elimination
- Loop Distribution
- Loop Interchange
- Loop Fusion
- Finish Elimination
- Loop Switching
- Loop Interchange
- Cleanup Optimizations

Cleanup Optimizations and Interprocedural Coarsening

- Redundant Next/Next-single Elimination.
- Loop chunking as a post-pass
- Copy propagation, dead-assignment elimination, unreachable-code elimination, loop fusion as post-pass.
- Interprocedural extensions to forall coarsening e.g.,

```plaintext
Interprocedural Loop interchange:
for (i : [1..n])
  foo();
void foo () {
  forall(point p:R)
  // n does not depend on p
  // R does not depend on i
  S;
} =>
forall(point p:R)
for (i: [1..n])
  foo();
void foo() {
  S;
}
```
Redundant Next/Next-Single Elimination (RNSE)

- A next statement is considered redundant if the task drops the corresponding phaser without accessing any shared state (updated by another task in the same phase) after the barrier call.
- A next single statement `{next S;}` can be replaced by `{next;S;}`, if multiple parallel instances of the statement S can be executed independent of each other.
- A next statement is considered redundant if it always precedes another barrier, and the two sets of tasks registered on the phasers of these barriers are identical.

Implementation and Evaluation
Experimental Evaluation of Forall Coarsening (CG, MG)

Summary (including other benchmarks):

<table>
<thead>
<tr>
<th>System</th>
<th>threads/cores</th>
<th>Unopt (Geo Mean)</th>
<th>Opt (GM)</th>
<th>Opt+RNSE (GM)</th>
</tr>
</thead>
<tbody>
<tr>
<td>UltraSPARC T2</td>
<td>128</td>
<td>10.7X</td>
<td>45.8X</td>
<td>48.5X</td>
</tr>
<tr>
<td>Xeon</td>
<td>16</td>
<td>3.44X</td>
<td>6.98X</td>
<td>7.27X</td>
</tr>
<tr>
<td>Power7</td>
<td>32</td>
<td>2.69X</td>
<td>16.4X</td>
<td>17.2X</td>
</tr>
</tbody>
</table>

Example of Illegal Forall Chunking

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

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

```c
1: delta = epsilon+1; iters = 0;
2: forall (point[j] : [1:n]) {
3:   for (point[j] : [jj:min(j)+d-1:n]) {
4:     while (delta > epsilon) {
7:       next single ( // barrier with single statement
8:       delta = diff.sum(); iters++;
9:       temp = newA; newA = oldA; oldA = temp;
} // next single
} // while
} // for
} // forall
```
Example of Legal Forall Chunking

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

```java
1: delta = epsilon+1; iters = 0;
2: forall (point[j]: [1:n1]) {
3:   while (delta > epsilon) {
6:     // sum and exchange
7:     next single {
8:       delta = diff.sum(); iters++;
9:       temp=newA; newA=oldA; oldA=temp;
7:     } // next single
6:   } // while
5: } // forall
```

Goal:
Correct chunking transformation to keep original semantics

Step 1: Strip mining
Double nested foreach loops

Step 2: Isolation of `next`
Legal combinations of
- Loop interchange
- Loop distribution
- Loop unswitching

Step 3: Serialization

Overview of Parallel Loop Chunking

**Step 1:**
Apply strip mining to the parallel loop

**Step 2:**
Isolate next synchronization statements
Legal combinations of Loop transformations

**Step 3:**
Serialize inner parallel loops

* i-foreach represents “inner foreach” that is created by compiler during transformation

---

Running Example

### Strip mining of the outermost foreach loop

// Original code
finish {
    ph = new phaser();
    foreach (point i: R) phased {
        for (int j = 0; j < m; j++) {
            S1;
            next;
            if (array[j] != 0) {
                for (int k = 0; k < l; k++) {
                    S2;
                    next;
                }
            }
        }
    }
}

// After Strip Mining
finish {
    ph = new phaser();
    foreach (point g: Ig(R)) phased {
        i-foreach (point i: le(R, g)) phased {
            for (int j = 0; j < m; j++) {
                S1;
                next;
                if (array[j] != 0) {
                    for (int k = 0; k < l; k++) {
                        S2;
                        next;
                    }
                }
            }
        }
    }
}

$Ig(R)$ = Iterator over multiple chunks of region $R$
$le(R, g)$ = Iterator over the elements of the chunk $g$
Running Example

Loop Interchange of i-foreach with for-j loop

// After Strip Mining
finish {
  ph = new phaser();
  foreach (point g: Ig(R)) phased {
    i-foreach (point i : le(R, g)) phased {
      for (int j = 0; j < m; j++) {
        S1;
        next;
        if (array[j] != 0) {
          for (int k = 0; k < l; k++) {
            S2;
            next;
          }
        }
      }
    }
  }
}

// After Loop Interchange
finish {
  ph = new phaser();
  foreach (point g: Ig(R)) phased {
    for (int j = 0; j < m; j++) {
      i-foreach (point i : le(R, g)) phased {
        S1;
        next;
        if (array[j] != 0) {
          for (int k = 0; k < l; k++) {
            S2;
            next;
          }
        }
      }
    }
  }
}

Running Example

Loop distribution of i-foreach

// After Loop Interchange
finish {
  ph = new phaser();
  foreach (point g: Ig(R)) phased {
    for (int j = 0; j < m; j++) {
      i-foreach (point i : le(R, g)) phased {
        S1;
        next;
        if (array[j] != 0) {
          for (int k = 0; k < l; k++) {
            S2;
            next;
          }
        }
      }
    }
  }
}

// After Loop Distribution
finish {
  ph = new phaser();
  foreach (point g: Ig(R)) phased {
    for (int j = 0; j < m; j++) {
      i-foreach (point i : le(R, g)) phased {
        S1;
        }
      }
    }
  }
}

if (array[j] != 0) {
  for (int k = 0; k < l; k++) {
    S2;
    next;
  }
}
}
Running Example

Loop unswtching of i-foreach with if-statement

// After Loop Distribution
finish {
    ph = new phaser();
    foreach (point g: Ig(R)) phased {
        for (int j = 0; j < m; j++) {
            i-foreach (point i : le(R, g)) phased {
                S1;
            }
            i-foreach (point i : le(R, g)) phased {
                next;
            }
            i-foreach (point i : le(R, g)) phased {
                if (array[j] != 0) {
                    for (int k = 0; k < l; k++) {
                        S2;
                    }
                }
            }
        }
    }
}

// After Loop Unswtching
finish {
    ph = new phaser();
    foreach (point g: Ig(R)) phased {
        for (int j = 0; j < m; j++) {
            i-foreach (point i : le(R, g)) phased {
                S1;
            }
            i-foreach (point i : le(R, g)) phased {
                next;
            }
            i-foreach (point i : le(R, g)) phased {
                if (array[j] != 0) {
                    i-foreach (point i : le(R, g)) phased {
                        for (int k = 0; k < l; k++) {
                            S2;
                        }
                    }
                }
            }
        }
    }
}

Running Example

Loop interchange and distribution (2nd)

// After Loop Unswtching
finish {
    ph = new phaser();
    foreach (point g: Ig(R)) phased {
        for (int j = 0; j < m; j++) {
            i-foreach (point i : le(R, g)) phased {
                S1;
            }
            i-foreach (point i : le(R, g)) phased {
                next;
            }
            if (array[j] != 0) {
                i-foreach (point i : le(R, g)) phased {
                    for (int k = 0; k < l; k++) {
                        S2;
                    }
                }
            }
        }
    }
}

// After Loop Interchange and Distribution
finish {
    ph = new phaser();
    foreach (point g: Ig(R)) phased {
        for (int j = 0; j < m; j++) {
            i-foreach (point i : le(R, g)) phased {
                S1;
            }
            i-foreach (point i : le(R, g)) phased {
                next;
            }
            if (array[j] != 0) {
                i-foreach (point i : le(R, g)) phased {
                    for (int k = 0; k < l; k++) {
                        S2;
                    }
                }
            }
        }
    }
}
Running Example

i-foreach serialization and next contraction

// After Loop Interchange and Distribution
finish {
ph = new phaser();
foreach (point g : Ig(R)) phased {
    for (int j = 0; j < m; j++) {
        i-foreach (point i : le(R, g)) phased {
            S1;
        }
        i-foreach (point i : le(R, g)) phased {
            next;
        }
    }
    if (array[j] != 0) {
        for (int k = 0; k < l; k++) {
            i-foreach (point i : le(R, g)) phased {
                S2;
            }
            i-foreach (point i : le(R, g)) phased {
                next;
            }
        }
    }
}
}

// After Serialization and Next Contraction
finish {
ph = new phaser();
foreach (point g : Ig(R)) phased {
    for (int j = 0; j < m; j++) {
        for (point i : le(R, g)) {
            S1;
        }
    }
    next;
    if (array[j] != 0) {
        for (int k = 0; k < l; k++) {
            for (point i : le(R, g)) {
                S2;
            }
        }
    }
}
}

Experimental Environment

- Loop chunking Framework
  - Implemented in Habanero-Java compiler with Soot-based Parallel Intermediate Representation
- System
  - 64-way (8-core x 8 threads/core) 1.2 GHz UltraSPARC T2 (Niagara 2)
  - 32 GB memory, running Solaris 10
  - Habanero-Java Work-Sharing Runtime
  - Java(TM) 2 Runtime Environment (build 1.5.0 12-b04) with Java Hot-Spot(TM) Server VM
- Benchmarks
  - Java Grande Forum Benchmarks
    - Crypt, FFT, LUFact, Series,SOR,SparseMatmult,Euler,MolDyn, MonteCarlo, RayTracer
    - HJ code with same parallelism as thread v1.0 (multithreaded version)
  - Nas Parallel Benchmarks 3.0
    - CG in HJ with same parallelism as the original
- Experimental variants
  - Serial Java
  - Parallel HJ w/o loop chunking
  - Parallel HJ with loop chunking implemented in Soot-based Parallel Intermediate Representation (PIR) Compiler Transformation Framework
Speedup with Auto Chunking on 8-core x 8-thread Niagara 2 SMP:
Java Grande Forum Benchmarks & Nas Parallel Benchmark CG

* Benchmarks with an asterisk have next operations within foreach loops
  - 2182x faster than no-chunk for crypt
  - 15.5x faster than no-chunk in geometric mean

```
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 ) {
        #pragma omp barrier
        if (j == 1) {
            delta = sum(diff); iters++;
            temp = newA; newA = oldA; oldA = temp;
        }
        #pragma omp barrier
    }
}
```

Why is this not a problem for chunking OpenMP parallel loops?

OpenMP prohibits synchronization inside a parallel loop

Unpredictable results on different platforms

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

V. Sarkar, PLDI Tutorial, June 2011
Another HPIR Optimization: Finish Elimination

- Goal: eliminate and/or reshape finish regions to reduce synchronization overhead and increase parallelism
- The impact of this optimization depends on the relative overhead of task termination with underlying runtime scheduling policy such as work-sharing or work-stealing.

BOTS Health Benchmark with Recursive Asyncs

```java
// 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:       // Conditional async
  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:    } // finish
        9:    } else {
            10:    final Iterator it=village.forward.iterator();
            11:    while (it.hasNext()) {
            12:        final Village v = (Village)it.next();
            13:        sim_village_par(v);
            14:    } // while
            15:    } // finish
            16:    ... ...
        17:    }
        18:    ... ...
    }
}

Performance Improvement due to Finish Elimination on 16-core Intel Xeon

- Improvement factor relative to the original parallel code
- Habanero-Java Work-sharing runtime
- Geometric mean on 16 cores = 2.88x
Outline of Tutorial

1) Overview of task-parallel languages
   - Cilk, OpenMP 3.0, Chapel, X10, Habanero-Java (HJ)
2) Optimizations of HJ programs at the High-level Parallel Intermediate Representation (HPIR) level
   - May-Happen-in-Parallel (MHP) analysis
   - Forall coarsening
   - Forall chunking
   - Finish elimination
3) Optimizations of HJ programs at the Middle-level (MPIR) and Low-level (LPIR) Parallel Intermediate Representations
   - Load elimination
   - Optimizations for work-stealing runtime schedulers
4) Communication optimizations of X10 programs on distributed-memory parallel machines
MPIR example: Load Elimination [BS09, Bar09]

- 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

Original Code

\[
\begin{align*}
p &:= \text{new Type1} \\
q &:= \text{new Type1} \\
&\vdots \\
p.x &:= p.x \\
q.x &:= \ldots \\
&\ldots \\
\end{align*}
\]

Transformed code

\[
\begin{align*}
p &:= \text{new Type1} \\
q &:= \text{new Type1} \\
&\vdots \\
T1 &:= p.x \\
p.x &:= T1 \\
q.x &:= \ldots \\
&\ldots \\
\end{align*}
\]

More details on Load Elimination optimization to follow later in tutorial
Example of Load Elimination Example in HJ

```c
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:   }
10:  foo()
11: }
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:   }
```

Can be replaced by a scalar

Can not be replaced by a scalar


Side-Effect Analysis

- Effects of function calls
  - What variables may be modified as side effects of a function call
- 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-Effect Representation

- Heap Array representation
  - Compile-time abstraction of runtime threads
  - Each object field \( x \) is abstracted using a distinct heap array, \( H^x \)
    - Each object field \( x \) is abstracted using a distinct heap
      - \( H^x \) represents all instances of field reference \( x \)
  - Memory load of \( a.x \) is represented as an memory use of \( H^a[a] \)
  - Memory store of \( a.x \) is represented as an memory write of \( H^a[a] \)
  - \( H^a[a] \) and \( H^b[b] \) are definitely same (DS) if value numbers of \( a \) and \( b \) are same

Example: Side-Effects for Parallel Constructs

```c
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:    }
10:  foo()
11: }
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: }

GMOD (bar) = \{H^r[z]\}
GREF (bar) = \{H^r[z]\}
```
Side-Effects for Parallel Constructs

- Method Level Side-Effects (GMOD, GREF)
  - GMOD and GREF sets to denote the set of heap arrays modified or referenced either directly or indirectly respectively.

\[
IMOD(p) = \{ H^a \exists s \in p, s \in \{PUTFIELD a.x, PUTSTATIC a.x \} \}
\]

\[
IREF(p) = \{ H^a \exists s \in p, s \in \{GETFIELD a.x, GETSTATIC a.x \} \}
\]

\[
GMOD(p) = IMOD(p) \cup \{GMOD(q)\}
\]

\[
GREF(p) = IREF(p) \cup \{GREF(q)\}
\]

- Finish Scope Level Side-Effect (FMOD, FREF)
  - Any async created within a finish scope must be completed before the statement after it is executed.
  - FMOD and FREF sets for a finish scope comprise of heap array accesses within the finish scope and field accesses made in async’s called within the finish scope.
  - Important for code motion around finish scope.

\[
FMOD(f) = \begin{cases} 
\bigcup_{s \in f, s \text{ invokes } q} \{GMOD(q) \cup EMOD(q)\} & \text{if } q \text{ is an async call} \\
\bigcup_{s \in f, s \text{ invokes } q} \{EMOD(q)\} & \text{otherwise}
\end{cases}
\]

\[
FREF(f) = \begin{cases} 
\bigcup_{s \in f, s \text{ invokes } q} \{GREF(q) \cup EREF(q)\} & \text{if } q \text{ is an async call} \\
\bigcup_{s \in f, s \text{ invokes } q} \{EREF(q)\} & \text{otherwise}
\end{cases}
\]
Side-Effects for Parallel Constructs

- **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

\[
EMOD(p) = \begin{cases} 
\bigcup_{s \in \text{static enums}} q \{ \neg F(s, p) \wedge (GMOD(q)) \} & \text{if } q \text{ is an async call} \\
\bigcup_{s \in \text{static enums}} q \{ \neg F(s, p) \wedge (EMOD(q)) \} & \text{otherwise}
\end{cases}
\]

\[
EREF(p) = \begin{cases} 
\bigcup_{s \in \text{static enums}} q \{ \neg F(s, p) \wedge (GREF(q)) \} & \text{if } q \text{ is an async call} \\
\bigcup_{s \in \text{static enums}} q \{ \neg F(s, p) \wedge (EREF(q)) \} & \text{otherwise}
\end{cases}
\]

---

Side-Effects for Parallel Constructs

- **Isolated Block Level Side-Effect (AMOD, AREF)**
  - allows mutual exclusion between asyncs within a single place
  - AMOD and AREF represent all the object fields modified and referenced within every isolated block (global summaries)
  - Important for code motion around isolated blocks
    - Strongly tied to underlying memory model

\[
AIMOD(p) = \{ \text{H} \in \text{a.x} \exists s \in p, I(s, p) \wedge s \in \{ \text{PUTFIELD a.x, PUTSTATIC a.x} \} \}
\]

\[
AIREF(p) = \{ \text{H} \in \text{a.x} \exists s \in p, I(s, p) \wedge s \in \{ \text{GETFIELD a.x, GETSTATIC a.x} \} \}
\]

\[
AGMOD(p) = AIMOD(p) \bigcup \{ I(s, p) \wedge GMOD(q) \}
\]

\[
AGREF(p) = AIREF(p) \bigcup \{ I(s, p) \wedge GREF(q) \}
\]

\[
AMOD = \bigcup_{p \in P} AGMOD(p)
\]

\[
AREF = \bigcup_{p \in P} AGREF(p)
\]
Example: Side-Effects of Parallel Constructs

1: void main() {
2:    p.x = ...
3:    s.w = ...
4:    finish // f1
5:        async //async_1
6:            p.x = ...
7:        isolated { q.y = ...; ... = q.y }
8:            ... = p.x
9:    }
10:    foo()
11: }
12:    ...
13:    ...
14: }
15: void foo() {
16:    async //async_2
17:        r.z = ...
18:        ...
19:        ...
20: }

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 (allows more optimization)

IC Memory Model Examples

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

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

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

Case 4
1: final A a = new A ()
2: a.f = ...
3: async { isolated if (...) a.x++ }
4: ... = a.f
Sequential Consistency [Lam97, AG96, Hil98]

- **SC** constrains all memory operations:
  - Write → Read
  - Write → Write
  - Read → Read, Write
- Simple model for reasoning about parallel programs
- But, intuitively reasonable reordering of memory operations in a uniprocessor may violate sequential consistency model
  - Modern microprocessors reorder operations all the time to obtain performance e.g., write buffers, overlapped writes, non-blocking reads...
  - Optimizing compilers perform code transformations that have the effect of reordering memory operations e.g., scalar replacement, register allocation, instruction scheduling, ...
  - A programmer may perform similar code transformations for software engineering reasons without realizing that they are changing the program’s semantics

[Lamport] “A multiprocessor system is sequentially consistent if the result of any execution is the same as if the operations of all processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by the program”
Weak Ordering

- Weak ordering:
  - Divide memory operations into data operations and synchronization operations
  - Synchronization operations act like a fence:
    - All data operations before synch in program order must complete before synch is executed
    - All data operations after synch in program order must wait for synch to complete
    - Synchs are performed in program order
  - Hardware implementation of fence: processor has counter that is incremented when data op is issued, and decremented when data op is completed

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

Experimental Setup

- Hardware
  - 16-core system that has four 2.40GHz quad-core Intel Xeon processors, 30GB of memory
- Operating System
  - Red Hat Linux (RHEL 5)
- Compiler and Runtime
  - Jikes RVM 3.0.0 with -X:ao:initial:compiler=opt, -X:irc:O0, PLOS_FRAC=0.4f
  - HJ work-sharing runtime with NUMBER_OF_LOCAL_PLACES set to 1 and INIT_THREADS_PER_PLACE set to number of workers
- Benchmark Set (5 largest HJ benchmarks)
  - Java Grande Forum (Moldyn, Montecarlo, RayTracer)
  - Nas Parallel Benchmarks (CG, MG)
  - specJBB (Hybrid X10+JUC constructs)
Speedup on 4 Quadcore Intel Xeon

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

Reduction in Dynamic Field Accesses

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th># getfield original</th>
<th># getfield after FKS Load elim.</th>
<th># getfield after FKS + TRANS Load elim.</th>
<th># getfield after PAR Load elim.</th>
<th># getfield after PAR + TRANS Load elim.</th>
<th>Impr. relative to Original (%)</th>
<th>Impr. Relative to FKS</th>
<th>Impr. Relative to FKS + TRANS</th>
</tr>
</thead>
<tbody>
<tr>
<td>CG-S</td>
<td>3.85E09</td>
<td>3.10E09</td>
<td>3.03E09</td>
<td>2.34E09</td>
<td>3.92E05</td>
<td>99.99%</td>
<td>99.99%</td>
<td>99.99%</td>
</tr>
<tr>
<td>MG-W</td>
<td>1.41E04</td>
<td>1.15E04</td>
<td>1.13E04</td>
<td>7.96E03</td>
<td>6.71E03</td>
<td>52.55%</td>
<td>41.72%</td>
<td>40.58%</td>
</tr>
<tr>
<td>MolDyn-B</td>
<td>1.19E10</td>
<td>7.91E09</td>
<td>5.82E09</td>
<td>4.91E09</td>
<td>3.11E09</td>
<td>73.89%</td>
<td>60.62%</td>
<td>46.49%</td>
</tr>
<tr>
<td>RayTracer-B</td>
<td>3.08E10</td>
<td>2.02E10</td>
<td>2.02E10</td>
<td>1.67E10</td>
<td>1.38E10</td>
<td>55.25%</td>
<td>31.93%</td>
<td>31.82%</td>
</tr>
<tr>
<td>Montecarlo-B</td>
<td>1.75E09</td>
<td>1.54E09</td>
<td>1.48E09</td>
<td>5.84E08</td>
<td>9.19E08</td>
<td>47.38%</td>
<td>40.48%</td>
<td>37.95%</td>
</tr>
<tr>
<td>specJBB</td>
<td>1.19E09</td>
<td>1.02E09</td>
<td>8.95E08</td>
<td>6.65E08</td>
<td>5.78E08</td>
<td>51.56%</td>
<td>43.19%</td>
<td>35.43%</td>
</tr>
</tbody>
</table>

FKS uses no side-effect analysis

Decrease in dynamic counts of getfield operations of upto ~99.99%
LPIR example: Frame-Store Optimization for Work-Stealing Runtime System

Source code

```java
int Foo(int x, int y) {
    int b;
    final int a = f1(x,y);
    frame = new FooFrame;
    frame.x = x;
    frame.y = y;
    frame.a = a;
    frame.b = b;
    async f2(a);
    b = f3(a,x);
    async f4(a);
    return f5(a,b);
}
```

LPIR-level pseudocode

```java
int Foo(int x, int y) {
    int b;
    final int a = f1(x,y);
    frame = new FooFrame;
    frame.x = x;
    frame.y = y;
    frame.a = a;
    frame.b = b;
    async f2(a);
    b = f3(a,x);
    frame.x = x;
    frame.y = y;
    frame.a = a;
    frame.b = b;
    async f4(a);
    return f5(a,b);
}
```

Frame-Store Optimization

- Live Variable Analysis
  - Use analysis results to remove frame-stores of variables that are not live at the continuation point
  - Should not include the uses of variables in the frame-store statements
  - Also applies to frame-stores of uninitialized locals
Frame-Store Optimization

```java
int Foo(int x, int y) {
  int b;
  final int a = f1(x,y);
  frame = new FooFrame;

  frame.x = x;
  frame.a = a;
  async f2(a);
  b = f3(a,x);

  frame.a = a;
  frame.b = b;
  async f4(a);

  return f5(a,b);
}
```

Frame-Store Optimization

- Available Expressions Analysis
  - Use analysis results to remove frame-stores of variables that have already been stored in the frame and have not been redefined since
  - Considers the uses of variables in the frame-store statements as evaluation of the trivial expression containing the value of that variable
  - No other uses of variable should be considered
Frame-Store Optimization for Work-Stealing with Work-First Policy (WS-WFP)

- % Increase in Code Size for WS-WFP (w/o Frame-Store Optimizations)
- % Increase in Code Size for WS-WFP (w/ Frame-Store Optimizations)

Frame-Store Optimization: Dynamic Frame-Store Counts (w/ and w/o Frame-Store Optimizations)
Outline of Tutorial

1) Overview of task-parallel languages
   - Cilk, OpenMP 3.0, Chapel, X10, Habanero-Java (HJ)

2) Optimizations of HJ programs at the High-level Parallel Intermediate Representation (HPIR) level
   - May-Happen-in-Parallel (MHP) analysis
   - Forall coarsening
   - Forall chunking
   - Finish elimination

3) Optimizations of HJ programs at the Middle-level (MPIR) and Low-level (LPIR) Parallel Intermediate Representations
   - Load elimination
   - Optimizations for work-stealing runtime schedulers

4) Communication optimizations of X10 programs on distributed-memory parallel machines

Distributed Object Model of X10

- Serialization/deserialization of data for remote activities created using at and async constructs
  - Objects
    - Only global instance fields (immutable) of an object and the transitive closure of the object graph are serialized and deserialized
    - Serialized objects contain remote references (RR) to the original objects
  - Structs and functions (closures)
    - All the data members and their transitive closures are serialized (since they are implicitly immutable)
Challenges

While X10 is more productive than other distributed-memory programs such as MPI and SPMD PGAS models such as UPC and Co-Array Fortran, it incurs high performance overhead while it is used in its full generality.

- A key source of communication overhead relates to the serialization performed on objects, structs, and closures
- A key source of synchronization overhead arises from lightweight tasks across places

Key Contributions

- We introduce high-level compiler optimizations to
  - reduce communication overheads
    - Scalar replacement for global variables and arrays
    - Class splitting
    - Loop splitting to separate local and remote communications
  - reduce synchronization overheads
    - Strip-mining of distributed loops
      - Scalar expansion
      - Async coalesce
Communication Optimization: Scalar Replacement for Global Arrays

val i:int = ...;  
val j:int = ...;  
val v:Array[int](1) = new Array[int](n);  
at (p) async {  
... v(i);  
... v(j);  
}

// Transformed Code  
val i:int = ...;  
val j:int = ...;  
val v:Array[int](1) = new Array[int](n);  
val v_i:int = v(i);  
val v_j:int= v(j);  
at (p) async {  
... v_i;  
... v_j;  
}

// Original Code
val c1:C = new C(2,3);  
val c2:C = new C(3,4);  
at (p) async {  
... c1.x ...;  
... c2.x ...;  
... c2.y ...;  
}

// Transformed Code
val c1:C = new C(2,3);  
val c2:C = new C(3,4);  
val c1_x = c1.x;  
val c2_x = c2.x;  
v_c2_y = c2.y;  
at (p) async {  
... c1_x ...;  
... c2_x ...;  
... c2_y ...;  
}
Communication Optimization: Class Splitting

class C {
  public var x:int;
  public var y:int;
  public var z:int;
}

val v:Array[C](1) = new Array[C](n);
...

at (p) async {
  for (i : R) {
    ... v(i).x;
    ... v(i).y;
  }
}

val v_x:Array[int](1) = new Array[int](n);
val v_y:Array[int](1) = new Array[int](n);
...

at (p) async {
  for (i : R) {
    ... v_x(i);
    ... v_y(i);
  }
}

No need to serialize the z field and the object header!

Implementation: X10 Compiler/Runtime System

The Native Back End is used in this work
**Implementation: Communication Optimization pass**

Communication optimization pass applied on Program Structure Tree (PST) before Java/C++ backend performs bottom-up traversal on the PST to apply the transformations.

---

**Experimental Setup**

- **X10 Benchmarks**
  - RandomAccess with per-node local table size of 4096 and number of updates as $4096 \times \text{MAX\_PLACES} \times 4$
  - Nqueens
  - Java Grande Forum MolDyn benchmark
  - FMM and PME from ANU chemistry simulation system (both simulated with 20,000 atoms)

- **X10 compiler and runtime**
  - X10 version 2.0.6 (Latest release is 2.2)

- **Platforms**
  - 128-node BlueGene/P (part of a 4096-node system)
  - 32-node Nehalem with Infiniband (part of a 90-node system)
  - 16-node Power7 with Infiniband (part of a 18-node system)
### Number of Serialized bytes (in MB) across places

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>64</th>
<th>128</th>
</tr>
</thead>
<tbody>
<tr>
<td>Randomaccess</td>
<td>unopt</td>
<td>2.26</td>
<td>9.66</td>
<td>35.53</td>
<td>131.78</td>
<td>500.54</td>
<td>1943.08</td>
</tr>
<tr>
<td></td>
<td>opt</td>
<td>.92</td>
<td>2.86</td>
<td>6.87</td>
<td>15.04</td>
<td>31.53</td>
<td>63.86</td>
</tr>
<tr>
<td>Nqueens</td>
<td>unopt</td>
<td>19.17</td>
<td>44.51</td>
<td>82.60</td>
<td>148.83</td>
<td>271.79</td>
<td>518.36</td>
</tr>
<tr>
<td></td>
<td>opt</td>
<td>.002</td>
<td>.01</td>
<td>.04</td>
<td>.15</td>
<td>.59</td>
<td>2.29</td>
</tr>
<tr>
<td>MolDyn</td>
<td>unopt</td>
<td>551.63</td>
<td>1166.19</td>
<td>2167.10</td>
<td>4106.19</td>
<td>8099.84</td>
<td>16749.49</td>
</tr>
<tr>
<td></td>
<td>opt</td>
<td>.39</td>
<td>1.21</td>
<td>2.93</td>
<td>6.85</td>
<td>17.14</td>
<td>52.17</td>
</tr>
</tbody>
</table>

Number of bytes communicated for both Nqueens and Moldyn are reduced significantly, thereby explaining the runtime benefits.

### Number of activities spawned across places

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>64</th>
<th>128</th>
</tr>
</thead>
<tbody>
<tr>
<td>Randomaccess</td>
<td>unopt</td>
<td>65,540</td>
<td>131,080</td>
<td>262,160</td>
<td>524,320</td>
<td>1,048,640</td>
<td>2,097,280</td>
</tr>
<tr>
<td></td>
<td>opt</td>
<td>65,542</td>
<td>131,084</td>
<td>262,168</td>
<td>524,336</td>
<td>1,048,672</td>
<td>2,097,344</td>
</tr>
<tr>
<td>Nqueens</td>
<td>unopt</td>
<td>73,714</td>
<td>73,716</td>
<td>73,720</td>
<td>73,728</td>
<td>73,744</td>
<td>73,776</td>
</tr>
<tr>
<td></td>
<td>opt</td>
<td>6</td>
<td>12</td>
<td>24</td>
<td>48</td>
<td>96</td>
<td>192</td>
</tr>
<tr>
<td>MolDyn</td>
<td>unopt</td>
<td>4,192,256</td>
<td>4,192,256</td>
<td>4,192,256</td>
<td>4,192,256</td>
<td>4,192,256</td>
<td>4,192,256</td>
</tr>
<tr>
<td></td>
<td>opt</td>
<td>3,457</td>
<td>10,374</td>
<td>24,220</td>
<td>51,960</td>
<td>107,632</td>
<td>219,744</td>
</tr>
<tr>
<td>ANU-FMM</td>
<td>unopt</td>
<td>26,454</td>
<td>27,864</td>
<td>30,448</td>
<td>34,168</td>
<td>40,600</td>
<td>50,768</td>
</tr>
<tr>
<td></td>
<td>opt</td>
<td>25,430</td>
<td>26,836</td>
<td>29,416</td>
<td>33,128</td>
<td>39,544</td>
<td>49,680</td>
</tr>
<tr>
<td>ANU-PME</td>
<td>unopt</td>
<td>809,262</td>
<td>809,790</td>
<td>810,441</td>
<td>811,291</td>
<td>813,250</td>
<td>816,381</td>
</tr>
<tr>
<td></td>
<td>opt</td>
<td>808,695</td>
<td>809,171</td>
<td>809,537</td>
<td>810,116</td>
<td>810,712</td>
<td>811,822</td>
</tr>
</tbody>
</table>

Both Nqueens and MolDyn show significant benefit in the reduction of number of activities created at remote places.
Experimental Results (MolDyn): Execution time in seconds

- MolDyn results:
  - BlueGene/P: 34.46X
  - Nehalem: 2.99X
  - Power7: 2.73X

Experimental Results (NQueens): Execution time in seconds

- NQueens results:
  - BlueGene/P: 26.33X
  - Nehalem: 3.01X
  - Power7: 1.367X
Experimental Results (ANU-FMM): Execution time in seconds

FMM results:
- BlueGene/P: 1.519X
- Nehalem: 1.221X
- Power7: 1.182X

Rice Habanero Multicore Software Project:
Enabling Technologies for Extreme Scale

**Parallel Applications**

1. **Portable execution model**
   - Lightweight asynchronous tasks and data transfers
     - `async`, `finish`, `asyncMemcpy`
   - Locality control for task and data distribution
     - *hierarchical place tree*
   - Mutual exclusion
     - *ownership-based isolation*
   - Collective, point-to-point, stream synchronization
     - *phasers*

2. **Two-level programming model**
   - Declarative Coordination Language for Domain Experts, CnC (Intel Concurrent Collections)
   - Task-Parallel Languages for Tuning Experts, Habanero-Java (from X10 v1.5) and Habanero-C

**Extreme Scale Platforms**

http://habanero.rice.edu
Projects under way in the Habanero Group (http://habanero.rice.edu)

- NSF Expeditions Center for Domain-Specific Computing (CDSC)
  - Collaboration with UCLA, UCSB, OSU, http://www.cdsc.ucla.edu
- Habanero Concurrent Collections (CnC)
  - http://habanero.rice.edu/cnc (includes link to download)
  - Collaboration with Intel, UCLA (derived from Intel CnC)
- Habanero Java (HJ)
  - http://habanero.rice.edu/hj (includes link to download)
  - Collaboration with IBM, PSU (HJ derived from IBM X10 v1.5)
- Habanero C/C++ (HC)
  - Collaboration with U. Delaware, MIT
- DARPA-funded Platform Aware Compilation Environment (PACE)
  - Collaboration with OSU, Stanford, ETI, TI
- SRC FCRP Multiscale Systems Center (MuSyC)

Concurrent Collections macro-dataflow model for Domain Experts

- Stealth approach: don’t tell domain experts that they have to learn functional programming …
- … instead, ask them to specify their program as a graph with steps as vertices and semantic ordering constraints as edges
  - Producer-consumer ordering (data dependence)
  - Parent-child ordering (control dependence)
- Step internals can be implemented in any language
- CDSC implements CnC model in Habanero-Java and Habanero-C for modeling and mapping stages respectively
The Habanero-Java & Habanero-C models for Tuning Experts

- Tuning experts need to map and tune domain experts’ CnC model (graph + steps) onto parallel systems
  - Exploit parallelism across and within steps
  - Optimize Locality, Data Movement, Load balancing, Scheduling, ..

- Habanero Approach: support a portable abstract execution model that supports high performance with high productivity
  1. Lightweight dynamic task creation & termination
  2. Locality control --- task and data distributions
  3. Mutual exclusion and isolation
  4. Collective and point-to-point synchronization
- Any sequential language can be extended with this model e.g., Habanero-Java, Habanero-C, Habanero-Scala

Some Potential Topics for Future Work

- Increased precision of array and pointer data flow analyses for task-parallel programs (including condition vectors)
- Locality optimizations for execution of task-parallel programs on parallel memory hierarchies (hierarchical places)
- Co-design of compile-time and runtime optimizations e.g., runtime selection of seq clauses and chunk sizes
- PRE extensions to Load Elimination in task-parallel programs
Acknowledgments: Habanero Team

- Faculty
  - Vivek Sarkar
- Senior Research Scientist
  - Michael Burke
- Research Scientists
  - Zoran Budimlić, Philippe Charles, Jun Shirako, Jisheng Zhao
- Research Programmer
  - Vincent Cavé
- Postdocs
  - Edwin Westbrook, Yonghong Yan,
- PhD Students
  - Sanjay Chatterjee, Shams Imam, Deepak Majeti, Raghavan Raman, Dragoș Sbîrlea, Kamal Sharma, Alina Simion, Sağnak Taştrîlar
- Undergraduate Students
  - Max Grossman, Jungwoo Lee, Jarred Payne
- Other collaborators at Rice
  - Rich Baraniuk, Corky Cartwright, Keith Cooper, Tim Harvey, John Mellor-Crummey, Erzsebet Merenyi, Krishna Palem, David Peixotto, Bill Scherer, Linda Torczon, Lin Zhong

Key References


