Intermediate Language Extensions for Parallelism

Jisheng Zhao
Dept. of Computer Science, Rice University
Houston, Texas, USA
jisheng.zhao@rice.edu

Vivek Sarkar
Dept. of Computer Science, Rice University
Houston, Texas, USA
vsarkar@rice.edu

Abstract

An Intermediate Language (IL) specifies a program at a level of abstraction that includes precise semantics for state updates and control flow, but leaves unspecified the low-level software and hardware mechanisms that will be used to implement the semantics. Past ILs have followed the von Neumann execution model by making sequential execution the default, and by supporting parallelism with runtime calls for lower-level mechanisms such as threads and locks. Now that the multicore trend is making parallelism the default execution model for all software, it behooves us as a community to study the fundamental requirements in parallel execution models and explore how they can be supported by first-class abstractions at the IL level.

In this paper, we introduce five key requirements for Parallel Intermediate Representations (PIRs): 1) Lightweight asynchronous tasks and communications, 2) Explicit locality, 3) Directed Synchronization with Dynamic Parallelism:, 4) Mutual Exclusion and Isolation with Dynamic Parallelism, and 5) Relaxed Exception semantics for Parallelism. We summarize the approach being taken in the Habanero Multicore Software Research project at Rice University to define a Parallel Intermediate Representation (PIR) to address these requirements. We discuss the basic issues of designing and implementing PIRs within the Habanero-Java (HJ) compilation framework that spans multiple levels of PIRs. By demonstrating several program optimizations developed in the HJ compilation framework, we show that this new PIR-based approach to compiler development brings robustness to the process of analyzing and optimizing parallel programs and is applicable to a wide range of task-parallelism programming models available today.

Categories and Subject Descriptors D.1.3 [*Concurrent Programming*]: Parallel programming

SPLASH'11 Workshops, October 23–24, 2011, Portland, Oregon, USA.

General Terms Language

Keywords Compiler, Parallelism, Program Analysis and Optimization

1. Introduction

The computer industry is at a major inflection point due to the end of a decades-long trend of exponentially increasing clock frequencies. It is widely agreed that parallelism in the form of multiple power-efficient cores must be exploited to compensate for this lack of frequency scaling. Unlike previous generations of hardware evolution, this shift towards manycore computing will have a profound impact on software. A number of task-parallel programming models have been developed in response to the multicore trend, including OpenMP 3.0 [25], Cilk [6], Java Concurrency [12], Chapel [17], X10 [8], Habanero-Java (HJ) [7], and Habanero-C (HC) [16].

An Intermediate Language (IL) specifies a program at a level of abstraction that includes precise semantics for state updates and control flow, but leaves unspecified the low-level software and hardware mechanisms that will be used to implement the semantics. Thus far, compilers for task-parallel languages have mostly piggy-backed on ILs for sequential languages that follow the von Neumann execution mode. In this approach, parallel constructs are translated to opaque runtime calls for lower-level mechanisms such as threads and locks, and it is generally expected that the compiler will not perform code optimizations across these runtime calls. However, this approach is both fragile and restrictive. It is fragile because a smart sequential compiler may attempt optimizations across these library calls that may be legal for a sequential program, but illegal for a parallel program. It is restrictive because some optimizations that are possible across parallel constructs (*e.g.,* load elimination [5]) may be ruled out because of the presence of opaque library calls. Now that the multicore trend is making parallelism the default execution model for all software, it behooves us as a community to study the fundamental requirements in parallel execution models and explore how they can be supported by first-class abstractions at the IL level.

In this paper, we introduce three levels of Parallel Intermediate Representations (PIRs)[1] motivated by different kinds of analyses and transformations. We summarize the approach being taken with these three levels of PIRs in the Habanero-Java (HJ) compilation framework that spans multiple levels of PIRs. By demonstrating several program optimizations developed in the HJ compilation framework, we show that this new PIR-based approach to compiler development brings robustness to the process of analyzing and optimizing parallel programs and is applicable to a wide range of task-parallelism programming models available today. To the best of our knowledge, this is the first design and implementation of a compiler IL that explicitly represents task-parallel constructs as first-class IL primitives.

The rest of the paper is organized as follows. Section 2 gives a brief introduction of the HJ language and its execution model, including parallel constructs for task creation, locality, synchronization, and mutual exclusion. Section 3 gives details of the design of the PIR. Section 4 discusses the implementation of the PIR approach in the HJ language. Section 5 discusses multiple program transformations at the PIR level. Finally, Section 6 discusses related work, and Section 7 contains our conclusions.

2. Background: Parallel Extensions in the Habanero-Java (HJ) language

In this section, we briefly summarize the main parallel constructs available in the Habanero-Java (HJ) language [7]. Since that is the task parallel language used in this work. However, the PIR approach can be applicable to other task parallel languages as well.

async: The statement "**async** ⟨*stmt*⟩" causes the parent task to create a new child task to execute ⟨*stmt*⟩ *asynchronously* (*i.e.,* before, after, or in parallel) with the remainder of the parent task. Figure 1 illustrates this concept by showing a code schema in which the parent task, T_0, uses an async construct to create a child task T_1. Thus, STMT1 in task T_1 can potentially execute in parallel with STMT2 in task T_0.

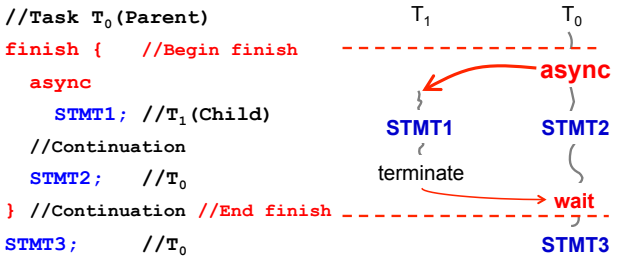

Figure 1. An example code schema with async and finish constructs

[1] We use the terms, "intermediate language (IL)" and "intermediate representation (IR)" interchangeably in this paper. A strict distinction would treat an IL as a persistent external representation of an IR defined as a set of internal data structures.

async is a powerful primitive because it can be used to enable any statement to execute as a parallel task, including for-loop iterations and method calls. In general, an HJ program can create an unbounded number of tasks at runtime. The HJ runtime system is responsible for scheduling these tasks on a fixed number of processors. It does so by creating a fixed number of *worker threads*, typically one worker per processor core or hardware context. These workers repeatedly pull work from one of those logical work queues when they are idle, and push work on queues when they generate more work. The work queue entries can include *asyncs* and *continuations*. An async is the creation of a new task, such as T_1 in Figure 1. A continuation represents a potential suspension point for a task, which (as shown in in Figure 1) can include the point after an async creation as well as the point following the end of a finish scope. Continuations are also referred to as *task-switching* points, because they are program points at which a worker may switch execution between different tasks.

As with Java threads, local variables are *private* to each task, whereas static and instance fields may be *shared* among tasks. An inner async is allowed to read a local variable declared in an outer scope. This semantics is similar to that of parameters in method calls — the value of the outer local variable is simply copied on entry to the async. However, an inner async is not permitted to modify a local variable declared in an outer scope. The ability to read non-final local variables in an outer scope is more general than the standard Java restriction that a method in an inner-class may only read a local variable in an outer scope if its declared to be final.

HJ also supports a seq clause for an async statement with the following syntax and semantics:

```
async seq(cond) <stmt> ≡
    if (cond) <stmt> else async <stmt>
```

The seq clause simplifies programmer-controlled serialization of task creation to deal with overheads. It is restricted to cases when no blocking operation such as phaser next operations and future get() operations is performed inside <stmt>. A key benefit of the seq clause, relative to programmer inserted threshold checks, is that it removes the burden on the programmer to specify <stmt> twice with the accompanying software engineering hazard of ensuring that the two copies remain consistent. In the future, the HJ system will explore approaches in which the compiler and/or runtime system can select the serialization condition automatically for async statements.

finish: finish is a generalized join operation. The statement "**finish** ⟨*stmt*⟩" causes the parent task to execute ⟨*stmt*⟩ and then wait until all async tasks created within ⟨*stmt*⟩ have completed, including transitively spawned tasks. Each dynamic instance T_A of an async task has a unique *Immediately Enclosing Finish* (IEF) instance F of a finish statement during program execution, where F is the innermost finish containing T_A [32]. There is an implicit

finish scope surrounding the body of main() so program execution will only end after all async tasks have completed.

Like async, finish is a powerful primitive because it can be wrapped around any statement thereby supporting modularity in parallel programming. The scopes of async and finish can span method boundaries in general. As an example, the finish statement in Figure 1 is used by task T_0 to ensure that child task T_1 has completed executing STMT1 before T_0 executes STMT3. If T_1 created a child async task, T_2 (a "grandchild" of T_0), T_0 will wait for both T_1 and T_2 to complete in the finish scope before executing STMT3. One approach to converting a sequential program into a parallel program is to insert async statements at points where the parallelism is desired, and then insert finish statements to ensure that the parallel version produces the same result as the sequential version.

Besides termination detection, the finish statement plays an important role with regard to exception semantics. If any async throws an exception, then its IEF statement throws a *MultiException* [8] formed from the collection of all exceptions thrown by all async's in the IEF.

future: HJ also includes support for async tasks with return values in the form of futures. The statement, "final future<T> f = async<T> Expr;" creates a new child task to evaluate Expr that is ready to execute immediately. In this case, f contains a "future handle" to the newly created task and the operation f.get() (also known as a *force* operation) can be performed to obtain the result of the future task. If the future task has not completed as yet, the task performing the f.get() operation blocks until the result of Expr becomes available. An important constraint in HJ is that *all variables of type* future<T> *must be declared with a* final *modifier*, thereby ensuring that the value of the reference cannot change after initialization. This rule ensures that no deadlock cycle can be created with future tasks. Finally, HJ also permits the creation of future tasks with void return type; in that case, the get() operation simply serves as a join on the future task.

phasers: The phaser construct [32] integrates collective and point-to-point synchronization by giving each task the option of registering with a phaser in *signal-only/wait-only* mode for producer/consumer synchronization or *signal-wait* mode for barrier synchronization. These properties, along with the generality of *dynamic parallelism*, *phase-ordering* and *deadlock-freedom* safety properties, distinguish phasers from synchronization constructs in past work including barriers [15] and X10's clocks [8]. The latest release of java.util.concurrent (j.u.c) in Java 7 includes Phaser synchronizer objects, which are derived in part [22] from the phaser construct in HJ. (The j.u.c. Phaser class only supports a subset of the functionality available in HJ phasers.)

In general, a task may be registered on multiple phasers, and a phaser may have multiple tasks registered on it. Three key phaser operations are:

• *new:* When a task A_i performs a new phaser() operation, it results in the creation of a new phaser *ph* such that A_i is registered with *ph* in the *signal-wait* mode (by default).

• *registration:* The statement, async phased (*ph1*⟨*mode1*⟩, *ph2*⟨*mode2*⟩, ...) ⟨*stmt*⟩, creates a child task that is registered on phaser *ph1* with *mode1*, phaser *ph2* with *mode2*, etc. The child tasks registrations must be subset of the parent task's registrations. async phased ⟨*stmt*⟩ simply propagates all of the parents phaser registrations to the child.

• *next:* The statement next is a synchronization operation that has the effect of advancing each phaser on which the invoking task A_i is registered to its next phase, thereby synchronizing all tasks registered on the same phaser. In addition, a next statement for phasers can optionally include a *single* statement, next {S}. This guarantees that the statement S is executed exactly once during the phase transition [39, 32].

foreach: The statement foreach (point p : R) S supports parallel iteration over all the points in region R by launching each iteration as a separate async. A foreach statement does not have an implicit finish (join) operation, but its termination can be ensured by enclosing it within a finish statement at an appropriate outer level. Further, any exceptions thrown by the spawned iterations are propagated to its IEF instance. Thus, we see that foreach (point p : R) S(p) is semantically equivalent to a combination of a sequential for loop with a local async construct for each iteration, which can be written as follows: for (point p : R) async {S(p)}.

forall: The forall construct is an enhancement of the foreach construct. The statement forall (point p : R) S supports parallel iteration over all the points in region R by launching each iteration as a separate async, and including an implicit finish to wait for all of the spawned asyncs to terminate.

Each dynamic instance of a forall statement also includes an implicit phaser object (let us call it ph) that is set up so that all iterations in the forall are registered on ph in *signal-wait* mode. One way to relate forall to foreach is to think of forall ⟨*stmt*⟩ as syntactic sugar for "ph=new phaser(); finish foreach phased (ph) ⟨*stmt*⟩". Since the scope of ph is limited to the implicit finish in the forall, the parent task will drop its registration on ph after all the forall iterations are created. The forall statement was designed for use as the common way to express parallel loops. However, programmers who need to perform finer-grained control over phaser registration for parallel loop iterations will find it more convenient to use foreach instead.

isolated: The HJ construct, isolated ⟨*stmt1*⟩, guarantees that each instance of ⟨*stmt1*⟩ will be performed in mutual

exclusion with all other potentially parallel *interfering* instances of `isolated` statements ⟨*stmt2*⟩. Two instances of `isolated` statements, ⟨*stmt1*⟩ and ⟨*stmt2*⟩, are said to interfere with each other if both access the same shared location, such that at least one of the accesses is a write. As advocated in [19], we use the `isolated` keyword instead of `atomic` to make explicit the fact that the construct supports weak isolation rather than strong atomicity. Commutative operations, such as updates to histogram tables or insertions into a shared data structure, are a natural fit for `isolated` blocks executed by multiple tasks in deterministic parallel programs.

The current HJ implementation takes a simple *single-lock* approach to implementing `isolated` statements, by treating each entry of an `isolated` statement as an *acquire()* operation on the lock, and each exit of an `isolated` statement as a *release()* operation on the lock. Though correct, this approach essentially implements `isolated` statements as *critical sections*, thereby serializing interfering and non-interfering `isolated` statement instances.

An alternate approach for implementing `isolated` statements being explored by the research community is *Transactional Memory* (TM) [19]. However, there is as yet no currently available practical TM approach in widespread use. Recently, a new implementation technique called *delegated isolation* [21] has been designed and prototyped for HJ `isolated` statements, and shown to be superior to both single-lock and TM approaches in many cases. We expect to include this technique in an HJ release in the near future.

places: The *place* construct in HJ provides a way for the programmer to specify affinity among `async` tasks. A place is an abstraction for a set of worker threads. When an HJ program is launched with the command, "`hj -places p:w`", a total of $p \times w$ worker threads are created with w workers per place. The places are numbered in the range $0 \ldots p-1$ and can be referenced in an HJ program, as described below. The number of places remains fixed during program execution; there is no construct to create a new place after the program is launched. This is consistent with other runtime systems, such as OpenMP, CUDA and MPI, that require the number of worker threads/processes to be specified when an application is launched. However, the management of individual worker threads within a place is not visible to an HJ program, giving the runtime system the freedom to create additional worker threads in a place, if needed, after starting with w workers per place.

The main benefit of using $p > 1$ places is that an optional `at` clause can be specified on an `async` statement or expression of the form, "`async at(`*place-expr*`) ...`", where *place-expr* is a place-valued expression. This clause dictates that the child `async` task can only be executed by a worker thread at the specified place. Data locality can be controlled by assigning two tasks with the same data affinity to execute in the same place. If the `at` clause is omitted, then the child task is scheduled by default to execute at the same place as its parent task. The main program task is assumed to start in place 0.

Thus, each task has a designated place. The value of a task's place can be retrieved by using the keyword, `here`. If a program only uses a single place, all `async` tasks just run at place 0 by default and there is no need to specify an `at` clause for any of them. The current release of HJ supports a flat partition of tasks into places. Support for hierarchical places [38] will be incorporated in a future release.

HJ Code Examples: We conclude this section with a brief discussion of two parallel programs written in HJ.

```
1   void sim_village_par(Village village) {
2     // Traverse village hierarchy
3     finish {
4       Iterator it=village.forward.iterator();
5       while (it.hasNext()) {
6         Village v = (Village)it.next();
7         async seq ((sim_level - village.level)
8                    >= bots_cutoff_value)
9           sim_village_par(v);
10      } // while
11      ... ...;
12    } // finish:
13    ... ...  }
```

Figure 2. Code fragment from BOTS Health benchmark written in HJ

Figure 2 shows a code fragment from the BOTS Health benchmark [10] rewritten in HJ. The `async seq` construct in line 7-9 executes the function, `sim_village_par(v)`, sequentially if the `seq` condition in line 7-8 is true, otherwise it creates a child task to invoke `sim_village_par(v)`. As a result, multiple child tasks created in multiple iterations can execute in parallel with the parent task. The parent task waits at the end of line 12 for all these child tasks to complete since the scope of the `finish` construct in this code fragment ends at line 12.

```
1  finish {
2      final int nproc = nthreads;
3      final phaser ph = new phaser();
4      foreach (point [proc]:[0:nproc-1])
5              phased (ph<hj.lang.phaserMode.SIG_WAIT>) {
6          for (int o = 0; o <= 1; o ++) {
7              int lim = (M-o) / 2;
8              SORrunIter(G, o, lim,
9                         proc, nproc);
10             next;
11         } } }
```

Figure 3. Code fragment from JGF SOR benchmark written in HJ

Figure 3 shows a fragment from a phaser-based HJ implementation of the JGF SOR benchmark [18]. Each iteration of the `foreach` loop is registered on the `phaser` object `ph`

in `sigwait` mode (line 5)[2]. The statement `next` (line 10) effectively performs a barrier operation among all iterations of the `foreach` loop. The procedure `SORrunIter` is the computation kernel. The `phaser` synchronization ensures that all data dependences in the algorithm are obeyed.

3. Parallel Intermediate Representation

This section discusses our approach to intermediate language extensions for parallelism. Figure 3 shows the structure of our `PIR` framework. The input is the abstract syntax tree (AST) nodes obtained from the language front-end, and the output is the target code that can in general be expressed in a source language, in bytecode, or machine code.

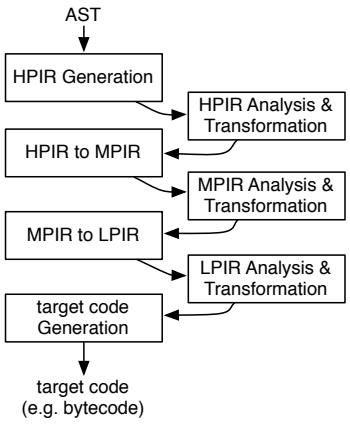

Figure 4. Overview of `PIR` Optimization Framework

Our framework contains three levels of `PIR` that are amenable to different kinds of analyses and transformations as follows:

1. The High level `PIR` (HPIR) has a hierarchical structure that's akin to the AST for many concurrency constructs, but flattens non-loop sequential constructs such as if-then-else statements. This level is suitable for efficient high level analyses such as May Happen in Parallel (MHP) [3], and to high level transformations such as `forall` chunking with phasers [33], and elimination of redundant `async` operations and strength reduction of termination `finish` operations to lighter-weight barrier (`next`) synchronizations [40].
2. The Middle level `PIR` (MPIR) lowers HPIR so that higher-level constructs like `foreach` and `forall` are translated to basic async-finish structures. This level is suitable for efficient data flow analyses in support of optimizations such as Load Elimination [5], where the flattened control flow simplifies data flow analysis compared to the HPIR, and the presence of runtime-independent `finish` and `async` operators simplifies analysis compared to the LPIR.
3. Low level `PIR` (LPIR) further lowers MPIR to expose runtime calls for concurrency constructs *e.g.,* runtime calls for work stealing. This level is suitable for optimization of runtime calls *e.g.,* optimization of frame-store operations in compiler support for work-stealing schedulers [13].

[2] An HJ `forall` construct would be more succinct since it contains an implicit `finish` and an implicit phaser. However, the explicit `finish`, `foreach` and phaser constructs in Figure 3 will make it easier to see the connection with later PIR examples.

```
1  FinishRegionEntry;
2    nproc = nthreads;
3    ph = new phaser();
4    specialInvoke ph.<init>();
5    i0 = nproc - 1;
6    ForeachRegionEntry iter(proc) region(0:i0)
7                  phasers(ph, SIG_WAIT>)
8      LoopRegionEntry iter(o) region([0:1])
9        if (o > 1) goto LoopRegionExit;
10       lim = (M-o) / 2;
11       staticInvoke SORrunIter(G, o,
12                                lim, proc, nproc);
13
14       NextOperation;
15       o = o + 1;
16       goto LoopRegionEntry;
17     LoopRegionExit;
18   ForeachRegionExit;
19 FinishRegionExit;
```

Figure 5. HPIR Code Example

Thus, the motivation for three levels of IR arises from the fact that different analyses and transformations can be performed more conveniently at different levels.

3.1 High Level PIR

The High Level `PIR` (HPIR) is based on the hierarchical Region Structure Graph (RSG) representation originally developed in the ASTI optimizer for IBM's XL Fortran compiler [29]. The RSG is composed of three major data structures described in the following subsections:

1. Region Structure Tree (RST)
2. Region Control Flow Graphs (RCFG's)
3. Region Dictionaries (RD's)

The backbone of the RSG is the Region Structure Tree (RST). The RST is a hierarchical structure with regions for loops and parallel source constructs in `PIR`. A Region Control Flow Graph (RCFG) and a Region Dictionary (RD) are maintained for each region. Together, the RST and the individual RCFG's and RD's comprise the Region Structure Graph representation.

3.1.1 Region Structure Tree (RST)

The Region Structure Tree (RST) represents the region nesting structure of the procedure being compiled. The regions are constructed for the following constructs : *finish*, *async*, *isolated*, *loop* (*for*, `foreach`, `forall`, *while*) and soot method/procedure. Each internal node (R-node) of the RST

represents a *single-entry region* of the input procedure, and each leaf node (L-node) of the RST corresponds to a single *IR statement*. Hierarchical nesting of regions is captured by the parent-child relation in the RST. The root node of the RST represents the entire procedure being compiled. We impose three key constraints on legal region structures in an RST:

1. **Tree Structure**

   The nesting structure for regions must form a single connected tree (specifically, the RST), and that there must be a one-to-one correspondence between leaf nodes in this tree and statements in the input IR. This constraint implies that if two regions $r1$ and $r2$ have a non-empty intersection, then it must be the case that either $r1 \subseteq r2$ or $r2 \subseteq r1$.

2. **Proper Containment**

   Each R-node must have at least one child in the RST that is an L-node. This implies that all regions are non-empty and that the region corresponding to a non-root R-node r must be properly contained within the region of its parent's node *parent*(r) (because it will not contain at least one L-node that is a child of *parent*(r)). Another consequence of this constraint is that there can be at most as many region nodes as there are `PIR` statements in the input procedure.

3. **Single-entry Regions**

   Each region must be single-entry. This is a natural consequence of using structured programming languages like Java. In the (rare) event that the input procedure contains irreducible control flow, then the entire irreducible subgraph must be included in a containing single-entry region.

An R-node serves as a useful anchor for all information related to the region corresponding to the R-node. In particular, pointers to the Region Control Flow Graph (RCFG), the Region Dictionary (RD) and Region Dependence Graph (RDG) can all be stored in the R-node for the region.

Note that there are no escaping `async`'s from a `finish` region, but a procedure's IR may have escaping `async`'s. So the procedure level region maintains a summary of the escaping `async`s *potentially* created within the procedure scope and (transitively) within all of its callees. Since this is a static summary, it is conservative for soundness and may include `async`'s that may never escape the procedure in any execution because of guard conditions that cannot be analyzed by the compiler.

Figure 5 shows an example HPIR fragment[3], obtained from the SOR HJ program introduced in Figure 3. The parallel constructs: `finish`, `for` loop, `foreach` loop are annotated by the region entry/exit labels (e.g. line 6 and 18 annotated the `foreach` loop region). These region labels are used to carry the information related to parallel constructs (e.g. loop iteration space, `phaser`, `place`), they are also used as the START/EXIT nodes in region control flow graph that will be introduced in the next subsection. Figure 6 shows the RST corresponding to the HPIR code listed in Figure 5.

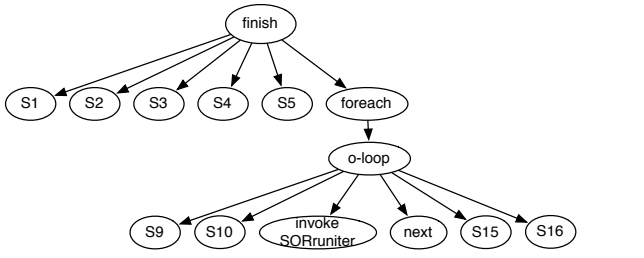

Figure 6. RST for HPIR example

3.1.2 Region Control Flow Graphs (RCFG's)

For each R-node, R, in the RST, we have a region-level control flow graph, *RCFG*(R), that defines the control flow for R's immediate children in the RST (the immediate children may be L-nodes or R-nodes). Note that we did not propose that acyclic control flow constructs like *if-then-else* and *switch* be mapped to separate regions in the RST. *RCFG*(R) must contain a node corresponding to each node that is a child of R. *RCFG*(R) also contains two pseudo nodes: START and EXIT. The pseudo nodes have the following interpretations:

- The START node is the source of all region entry branches. Since R must be a single-entry region, all LCFG edges from START have the same destination: the region's entry node (which must be a leaf node in the LST).
- The EXIT node is the target of all region exit branches.

As mentioned in the previous section, the START/EXIT nodes are represented as region entry/exit labels. These two labels also annotate the region scope and carry object-based parallel constructs (*e.g.,* `phaser()`, `place()`) that are related to the current region. For example, line 1 and 19 in Figure 5 show the START/EXIT node for the `finish` region, and they are reflected as `finish-entry` and `finish-exit` in Figure 7. Similarly, line 8 and 17 are for the `o-loop` region, corresponding to `loop-entry` and `loop-exit` nodes.

An edge e from X to Y in *RCFG*(R) represents normal or exceptional control flow in the current region. If edge e is a conditional branch (i.e., if there are multiple outgoing edges from X in *RCFG*(R)), then it also carries a *label* of the form (S, C) identifying condition C in IL statement S as the *branch condition* that enabled execution to flow along e.

For each control flow exit from region R, an edge is inserted in R's RCFG with target EXIT, and another edge is inserted in the RCFG of R's LST parent from R to the exit destination. If multiple regions are being exited (as in a `break`

[3] This `PIR` implementation is based on the Soot JIMPLE IR, which is discussed in next section.

statement or when an exception is thrown), then additional outer-region edges need to be inserted in enclosing regions.

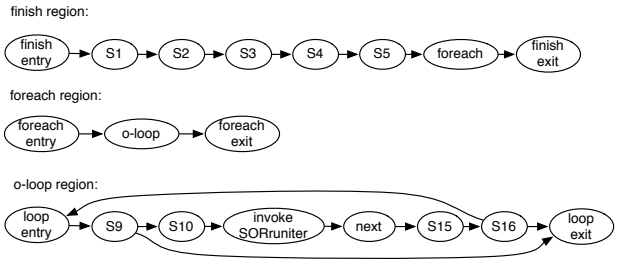

Figure 7. RCFG for HPIR example

3.1.3 Region Dictionaries (RD's)

For each R-node, R, in RST, we have a region dictionary, *RD*(R), that stores the *summary references* (exposed references):

- If *RD*(R) contains at least one exposed def of a variable V and R' = *parent*(R), then include a single *summary def* of variable V in *RD*(R'), and indicate if it is a *may-def* or a *must-def*.
- If *RD*(R) includes at least one exposed use of variable V and R' = *parent*(R), then include a single *summary use* of variable V in *RD*(R').

Summary references are only stored for local variables in RD's, They have the potential of propagating upwards in the RST when performing incremental re-analysis. However, we observe that the upward propagation only occurs when the first use/def is added to the child region or when the last use/def is removed from the child region. Hence, we expect the amortized overhead of upward propagation to be low when multiple references to a variable are inserted or deleted. In general, the dictionary for any single region will have a smaller number of references than a flat dictionary for the entire procedure, and hence should provide more efficient support for incremental reanalysis for a single region.

3.2 Middle Level PIR

The middle level `PIR` (MPIR) is a flat IR, with labels that designate the boundaries of high level constructs from HPIR that are lowered to MPIR *e.g.,* `async`, `finish`, and `isolated` constructs. Other high level constructs are lowered into combinations of these constructs and normal IR, e.g. `foreach` is translated to a for loop structure with an `async` loop body. Figure 8 gives an example of the MPIR code lowered from the HPIR shown in Figure 5.

MPIR inherits the region labels created at the HPIR level. MPIR also import two type edges that can help side-effect analysis for parallelism:

- *Happens-Before (HB):* a *directed* edge is added from a source statement S to a destination statement D if some instances of D may/must wait before starting for an instance of S to complete. HB edges are similar to *synchronization edges* in Parallel Programming Graphs [28, 31, 30] HB edges are labeled with *context* to identify the instances for which the happens-before relationship holds. They are also labeled as "may" or "must"edges.

  HB edges may come from multiple sources:

  - Edges for `finish` operations: an edge from a region exit label for an `async` and parallel loops (e.g. `ForEachRegionExit`) to its matching `FinishRegionExit`. (There will always be a unique intra-procedural `ExitFinishStmt` for a given region exit label, but there may be multiple potential inter-procedural `FinishRegionExit`'s for methods with escaping `async`.);
  - Edges for `phaser` operations: `next`, `signal`, `wait`, and `next-single` as outlined in [32].
- *Mutual-Exclusion (ME):* an *undirected edge* is added between two *isolated* statements that may execute in parallel. Context information can be used to identify which execution instances of the statements participate in the mutual exclusion relationship, as outlined in [3].

```
1  FinishRegionEntry;
2    nproc = nthreads;
3    ph = new phaser();
4    specialInvoke ph.<init>();
5    i0 = nproc - 1;
6    proc = 0;
7  entry_0:
8    if (proc > i0) goto exit_0;
9    AsyncRegionEntry phasers(ph, SIG_WAIT>)
10     o = 0;
11 entry_1:
12     if (o > 1) goto exit_1;
13     lim = (M-o) / 2;
14     staticInvoke SORrunIter(G, o,
15                                lim, proc, nproc);
16
17     NextOperation;
18     o = o + 1;
19     goto entry_1;
20 exit_1:
21   AsyncRegionExit;
22   proc = proc + 1;
23   goto entry_0;
24 exit_0:
25 FinishRegionExit;
```

Figure 8. MPIR Code Example

3.3 Low Level PIR

The Low Level `PIR` (LPIR) lowers all parallel constructs to combinations of runtime APIs and standard IR statements, thereby generating what appears to be standard sequential code. Figure 9 shows the LPIR code obtained from the MPIR code shown in Figure 8. The `async` closure in Figure 8 was extracted out-of-line as a new `Activity` class, whose `runHjTask` method contains the code from the `async`

```
act = staticInvoke hj.runtime.getCurrentActivity();
virtualInvoke act.startFinish();
  nproc = nthreads;
  ph = new phaser();
  specialInvoke ph.<init>();
  i0 = nproc - 1;
  proc = 0;
entry_0:
  if (proc > i0) goto exit_0;
  a0 = new Activity0;
  specialInvoke a0.<init>(ph, this, proc, nproc);
  act = staticInvoke hj.runtime.getCurrentActivity();
  place = virtualInvoke act.getPlace();
  virtualInvoke place.runAsync(a0);
  proc = proc + 1;
  goto entry_0;
exit_0:
act = staticInvoke hj.runtime.getCurrentActivity();
virtualInvoke act.stopFinish();
... ...
// Activity class
public class Activity0 {
  public phaser ph;
  public Sor thisobj;
  public int proc;
  public int nproc;

  // async closure
  public void runHjTask() {
    o = 0;
entry_1:
    if (o > 1) goto exit_1;
    lim = (thisobj.M-o) / 2;
    staticInvoke SORrunIter(thisobj.G, o,
                            lim, proc, nproc);

    virtualInvoke ph.doNext();
    o = o + 1;
    goto entry_1;
exit_1:
  }

  public void <init>(phaser, Sor, int, int) {
    this.ph = @param0;
    this.thisobj = @param1;
    this.proc = @param2;
    this.nproc = @param3;
  }
}
```

Figure 9. LPIR Code Example

body. The task spawn, `phaser` signal-wait and start/stop `finish` operations have all been translated to calls to the Habanero-Java runtime system.

For different runtime library that has different implementation, the LPIR translation has to generate corresponding API call and code layout, thus LPIR is runtime dependent. Since LPIR has same feature as the target language's IR, the optimizations related to the parallel runtime system are applied at this level.

4. Implementation

In this section, we describe an implementation of the `PIR` structure introduced in Section 3. This implementation was done for the Habanero-Java (HJ) language.

4.1 HJ Compiler and Runtime

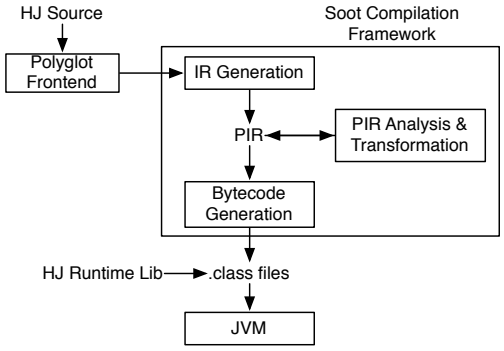

Figure 10. HJ Compiler Structure

As shown in Figure 4.1, the `HJ` compiler is primarily composed of two software components:

- The Polyglot [24] front-end that takes `HJ` source code and translates it to an AST representation;
- The Soot compilation framework [34] that includes: an HJ extension to translate the Polyglot AST to JIMPLE with our `PIR` extensions; multi-level `PIR` translation, analysis and transformations; and, generation of verifiable output Java classfiles that can run on a standard JVM with the `HJ` runtime library.

As mentioned in Section 3, the LPIR includes runtime-dependent code generation with calls to the `HJ` runtime library [7]. In `HJ`, the LPIR has to support code generation for multiple runtime scheduling policies *e.g.,* work-sharing (WSH), work-stealing with help-first (WST-HF) and work-stealing with work-first (WST-WF) policies [13, 14], as shown shown in Figure 11.

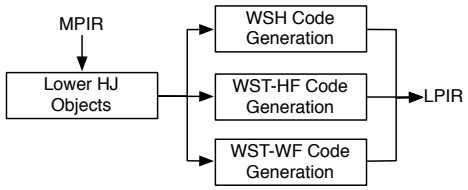

Figure 11. Runtime Dependent Code Generation

4.2 PIR Extensions

The HJ implementation of `PIR` is an extension of Soot's JIMPLE IR, which is a 3-address IR for expressing Java program. A detailed description of Soot and its IRs can be found in [34]. We focus this summary of JIMPLE on the `JimpleBody` structure which is used to represent a

method body. Soot also includes additional data structures for class, field, and method declarations, as well as a framework for intra-method and whole-program data flow analyses and points-to-analysis. A `JimpleBody` contains references to three `Chain`'s (lists) — a chain of *local variables*, a chain of *traps* (exception handlers), and a chain of *units* (statement blocks). Transformations can be performed on JIMPLE by creating *packs* of `BodyTransformer` objects,

JIMPLE is a stack-less typed IR with blocks of 3-address statements connected by a control flow graph. It is similar in content to the HIR (High level IR) used in Jikes RVM [11]. Statement operands can include local variables and heap locations. JIMPLE statements are classified as follows:

- **Core statements** include `AssignStmt`, `IdentityStmt`, and `NopStmt`. `IdentityStmt`'s are copy statements with special operands for parameters and the "`this`" reference as variables's. e.g., `r0 := @this; i1 := @parameter0;`
- **Intraprocedural control-flow statements** include `IfStmt`, `GotoStmt`, `TableSwitchStmt`, `LookupSwitchStmt`. (There is also a `RetStmt` for JSR return bytecode instructions, but it is not used when generating bytecode from Java or `HJ` programs.) Note that there is no explicit *label* statement; instead, control flow targets are implicitly represented in the control flow edges used to connect statement blocks. However, labels are automatically generated when pretty-printing JIMPLE.
- **Interprocedural control-flow statements** include `InvokeStmt`, `ReturnStmt`, and `ReturnVoidStmt`. A *new* statement is translated to two statements in JIMPLE, an `AssignStmt` for object instantiation with a `new` operator, and an `InvokeStmt` for the constructor.
- **Exception control-flow statements** include `ThrowStmt`.

The following subsections summarize four categories of extensions to JIMPLE to obtain an implementation of a `PIR` for Habanero-Java.

4.2.1 Closure-based Parallel Constructs

As shown in Figure 5 and 8, all closure-based parallel constructs (*e.g.,* `foreach` loops) should provide region entry/exit labels to identify the closure (*e.g.,* `foreachRegionEntry`, `foreachRegionExit`). These region entry/exit labels are currently extended with implicit side-effects to ensure that compiler transformations can not eliminate or reorder them. Figure 12 lists the class hierarchy of the region entry/exit objects for all closure-based parallel constructs in our implementation.

The region entry label also carries the list of relevant object based constructs, *e.g.,* `index-region` and `phaser` objects (see lines 6–8 in Figure 5).

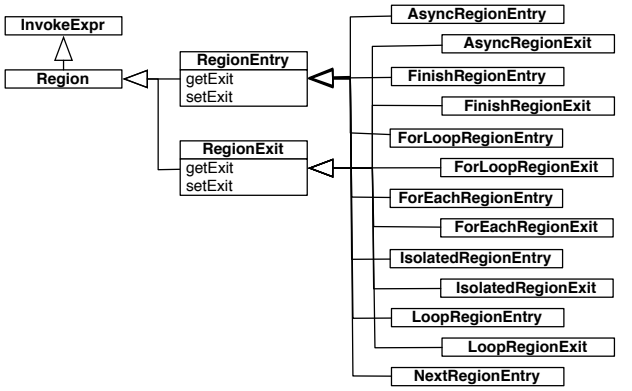

Figure 12. Class Hierarchy of Region Entry/Exit in PIR

```
1  public interface RSTNode {
2      // get statement
3      public Stmt getNodeStmt();
4
5      // maintation RST
6      public boolean hasSubNode(RSTNode node);
7      public void addSubNode(RSTNode node);
8      public RSTNode getParent();
9      public void setParent(RSTNode parent);
10     public boolean hasSubNode(RSTNode node);
11
12     // get/set RCFG
13     public UnitGraph getUnitGraph();
14     public void setUnitGraph(UnitGraph unitGraph);
15
16     // get/set Region Dictionary
17     public Set<Local> collectDefs(Body methodBody);
18     public Set<Local> collectUses(Body methodBody);
19
20     // check node types
21     public boolean isMethodNode();
22     public boolean isRegionNode();
23     public boolean isForEachNode();
24     public boolean isForLoopNode();
25     public boolean isFinishNode();
26     public boolean isAsyncNode();
27 ...
28 }
```

Figure 13. RSTNode Interface in PIR

4.2.2 Hierarchical Representation

The basic element of the hierarchical HPIR representation is an internal R-node. In our HPIR implementation, we use *RSTNode* objects to represent both R-nodes and L-nodes in the RST and RCFGs. Each *RSTNode* instance contains:

1. A reference to the *statement* that is either region entry label (for an R-node) or normal JIMPLE IR (for an L-node);
2. A reference to the region dictionary that contains both a *use set* and a *def set*;
3. A reference to the region's RCFG, implemented as a control flow graph object in Soot.

Figure 13 shows the *RSTNode* interface with appropriate accessor and modifier methods.

```
1  public interface Phaser extends HjObject {
2      public Value getModeValue();
3  }
4
5  // implementation
6  public class Phaser_c extends HjObject_c
7                        implements Phaser {
8      public Value getModeValue() {
9          return this.phaserModeBox.getValue();
10     }
11 ...
12 }
```

Figure 14. Phaser Object in PIR

4.2.3 Object-based Parallel Constructs

The parallel constructs related to synchronization and locality control are object-based *e.g.,* `phaser` and `place`. These constructs are implemented as implementations of the `HjObject` interface and `HjObject_c` class that can encapsulate the object reference and its attribute. For example, Figure 14 gives the implementation of the `phaser` interface and class for our `PIR` that maintains the `phaser` object reference and its phaser mode captured as an attribute.

4.2.4 Operation-based Parallel Constructs

Operation-based parallel constructs are those that perform operations on object-based parallel constructs, *e.g.,* signal, wait, and signal-wait operations that can be performed on phasers.. These constructs are represented as an entry-only region, (see example in line 14 in Figure 5 and line 17 in Figure 8).

5. PIR Applications

The previous sections summarized the design and implementation of three different levels of `PIR`. In this section, we briefly describe our experiences with different clients in the HJ compiler, for the HPIR, MPIR and LPIR levels.

5.1 High Level PIR

The HPIR provides rich information on the structure of the input parallel program. By leveraging the RST, the compiler can easily identify context information for all parallel constructs in the input program. We summarize three analyses and transformations that all leverage region structure information available in the HPIR, and have been implemented in the HJ compiler.

5.1.1 May-Happen-In-Parallel (MHP) Analysis

An efficient MHP analysis for the `X10` language[4] in [3]. To check if two statements, S_1 and S_2, may execute in parallel, this algorithm walks up the RST from S_1 and S_2 respectively, while terminating at their Least Common Ancestor (LCA). The MHP determination then follows from checking if S_1 is contained in an exposed `async` in the LCA. Details can be found in [3].

[4] We re-implemented this algorithm for `HJ`, with extensions to support interprocedural analysis.

5.1.2 Chunking Parallel Loops

An approach to chunking parallel loops with barrier synchronization operations was introduced in [33] , while also reducing/eliminating the overhead of task creation as much as possible. The transformation algorithm is divided into two stages:

1. Identify the parallel loops (`foreach` and `forall`) by traversing the RST;
2. Given the parallel loop as an entry point, apply loop transformation rules in a top-down manner to enlarge the granularity of parallelism.

Details can be found in [33] .

5.1.3 SPMDization

A new methodology for reducing the overhead of both task creation and termination by SPMDizing parallel loops (*e.g.,* `forall` or `finish` + `foreach`) was introduced in [40]. The algorithm encountered the RST in a bottom-up manner; for each parallel loop, it applies transformation rules in a top-down manner to expose redundant termination operations and enlarge the granularity of parallelism. At each level of the RST, the algorithm also attempts loop fusion, wherever it is legal to do so.

5.2 Middle Level PIR

The MPIR creates a flat `PIR` structure with labels for basic parallel constructs, including task creation, termination, and synchronization. It also maintains the *happens-before* and *mutual-exclusion* edges that can assist program analysis. Examples of MPIR-level compiler analysis and transformations are discussed below:

5.2.1 Load Elimination

An inter-procedural load elimination algorithm that can optimize redundant load operations in parallel programs with `async`, `finish` and `isolated` constructs was introduced in [5]. The basic approach is to perform side-effect analysis among all tasks that run in parallel or have a mutual exclusion relationship via the `isolated` construct. This algorithm checks each `async` closure and procedure with the assistance of happens-before and mutual exclusion information to identify potential side-effects and verify the safeness of load elimination.

5.2.2 Delegated Isolation

A novel execution model (called `AIDA`) for mutual exclusion in parallel programs was introduced in [21]. It performs frequent, irregular accesses to pointer-based shared data structures. A key compiler transformation in this approach is to insert object ownership instructions into `isolated` regions to enable the `isolated` code region to be executed

speculatively. This work is done at the MPIR level, which provides precise information for `isolated`, `async` and `finish` regions.

5.2.3 Data Race Detection

The ESP-bags algorithm [27] is based on instrumenting load/store operations within `async`, `finish` and `isolated` regions. Similarly, the *Permission Region* [36] approach uses `PIR` support to insert ownership verification code automatically and to avoid false positive by analyzing the `finish`, `async` regions. All of these instrumentations are performed at the MPIR level, and then subsequently optimized to eliminate redundant checks.

5.3 Low Level PIR

The LPIR has the same structure as a traditional sequential IR, since all parallel constructs are lowered to a combination of runtime API calls and sequential IR statements. In [26], Raman et. al discussed program optimizations related to code generation for work-stealing runtime systems. This work includes optimizing the context frame fields by eliminating redundant variables that were not accessed within tasks, and applying object inlining for frame objects so as to reduce the overhead of task spawning.

6. Related Work

In this section, we briefly summarize relevant past work on IR extensions to support analysis and transformation of explicitly parallel programs.

The Parallel Program Graph (PPG) was described in [31]. Compared with a sequential control flow graph, the PPG introduces an *mgoto control edge* for task creation, and a *synchronization edge* for directed task synchronization. In [30], it was shown how the PPG can be used to perform a reaching-def analysis for explicitly parallel programs.

The Concurrent SSA (CSSA) representation [20] introduced by Lee et. al is an analytical framework that represents parallel programs written using `cobegin`/`coend` constructs, along with event objects to perform inter-thread synchronization. A key restriction in this work is that loops are not permitted to contain parallel constructs, though a single sequential task may contain loops. The CSSA is built upon the concurrent CFG that is similar to PPG and includes conflict edges for interfering memory accesses that may be performed by distinct threads. The CSSA extended standard scalar SSA form [9] by adding special π functions to reflect the race conditions introduced by the conflict edges. In [23], Novillo further extended CSSA by supporting mutex edges that can help identify critical sections among parallel threads for more precise analysis of parallel program with mutual exclusion.

The `X10` [37] compilation system employs both the Polyglot front-end [24] and the WALA library [35] to perform program analysis for `X10` programs. Polyglot-level analysis and transformation is performed in a manner akin to HPIR, such as the communication optimizations performed in [4]. In contrast, the WALA-level analysis is performed in a manner akin to MPIR since WALA also represents regular statements at a level similar to a three-address form of Java bytecodes. Unlike the Soot JIMPLE IR (which supports both analyses and transformations), the WALA library only supports program analysis (and no transformations).

In general, much of the past related work either extended control flow and dataflow analyses for parallel programs or modified sequential approaches to perform program analysis in a restricted manner. In contrast, our framework introduced three levels of `PIR` (as discussed earlier) which span both hierarchical and flat representations so as to provide robust support for a wide range of program analyses and transformations.

7. Conclusions

In this paper, we introduced three levels of Parallel Intermediate Representations (`PIR`s) motivated by different kinds of analyses and transformations. We summarized the approach being taken with these three levels of `PIR`s in the Habanero-Java (HJ) compilation framework that spans multiple levels of `PIR`s. By demonstrating several program optimizations developed in the HJ compilation framework, we showed that this new `PIR`-based approach to compiler development brings robustness to the process of analyzing and optimizing parallel programs and is applicable to a wide range of task-parallel programming models available today. To the best of our knowledge, this is the first design and implementation of a compiler IL that explicitly represents task-parallel constructs as first-class IL primitives. Directions for future work include incorporating additional analyses and transformations in our `PIR` framework, as well as exploring a `PIR` implementation for C/C++ programs with the HPIR level implemented in Rose [2] and the MPIR and LPIR levels implemented in LLVM [1].

Acknowledgments

This work was supported in part by the National Science Foundation under the HECURA program, award number CCF-0833166. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect those of the National Science Foundation. We also gratefully acknowledge support from an IBM Open Collaborative Faculty Award for X10 optimizations, and would like to especially thank David Grove, Igor Peshansky, and Vijay Saraswat from IBM for early discussions on PIR design.

References

[1] The LLVM compiler infrastructure. http://llvm.org/.

[2] The ROSE open source compiler infrastructure. http://rosecompiler.org/.

[3] Shivali Agarwal et al. May-happen-in-parallel analysis of X10 programs. *PPoPP'97*, 1997.

[4] Rajkishore Barik et al. Communication optimizations for distributed-memory X10 programs. In *IPDPS '11*. IEEE Computer Society, 2011.

[5] Rajkishore Barik and Vivek Sarkar. Interprocedural load elimination for dynamic optimization of parallel programs. In *PACT'09*. IEEE Computer Society, 2009.

[6] R. D. Blumofe et al. CILK: An efficient multithreaded runtime system. *PPoPP'95*, pages 207–216, July 1995.

[7] Vincent Cavé, Jisheng Zhao, Jun Shirako, and Vivek Sarkar. Habanero-Java: the New Adventures of Old X10. In *PPPJ'11*, 2011.

[8] P. Charles et al. X10: an object-oriented approach to non-uniform cluster computing. In *OOPSLA*, NY, USA, 2005.

[9] Ron Cytron et al. An Efficient Method for Computing Static Single Assignment Form. *POPL'89*, January 1989.

[10] Alejandro Duran et al. Barcelona OpenMP tasks suite: A set of benchmarks targeting the exploitation of task parallelism in openmp. In *ICPP'09*, pages 124–131, 2009.

[11] B. Alpern et al. The jikes research virtual machine project: Building an open-source research community. *IBM Systems Journal special issue on Open Source Software*, 44(2), 2005. (see also http://jikesrvm.org).

[12] B. Goetz. *Java Concurrency In Practice*. Addison-Wesley, 2007.

[13] Yi Guo et al. Work-first and help-first scheduling policies for async-finish task parallelism. In *IPDPS'09*, 2009.

[14] Yi Guo et al. Slaw: a scalable locality-aware adaptive work-stealing scheduler for multi-core systems. In *IPDPS'10*, GA, USA, 2010.

[15] R. Gupta. The fuzzy barrier: a mechanism for high speed synchronization of processors. In *ASPLOS'89*, pages 54–63, New York, USA, 1989. ACM.

[16] Habanero c project web page. https://wiki.rice.edu/confluence/display/HABANERO/Habanero-C, January 2009.

[17] Cray Inc. The Chapel language specification version 0.4. Technical report, Cray Inc., February 2005.

[18] The Java Grande Forum benchmark suite. http://www.epcc.ed.ac.uk/javagrande/javag.html.

[19] James R. Larus and Ravi Rajwar. *Transactional Memory*. Morgan and Claypool, 2006.

[20] Jaejin Lee, Samuel P. Midkiff, and David A. Padua. Concurrent static single assignment form and constant propagation for explicitly parallel programs. In *PPoPP'97*, pages 114–130. ACM, 1997.

[21] Roberto Lublinerman, Jisheng Zhao, Zoran Budimlic, Swarat Chaudhuri, and Vivek Sarkar. Delegated Isolation. In *Proceedings of OOPSLA 2011*, 2011.

[22] Alex Miller. Set your Java 7 Phasers to stun. http://tech.puredanger.com/2008/07/08/java7-phasers/, 2008.

[23] Diego Novillo, Ron Unrau, and Jonathan Schaeffer. Concurrent SSA form in the presence of mutual exclusion. In *ICPP'98*, pages 356–364, 1998.

[24] Nathaniel Nystrom, Michael R. Clarkson, and Andrew C. Myers. Polyglot: An extensible compiler framework for Java. In *CC'03*, pages 1380–152, April 2003.

[25] OpenMP Application Program Interface, version 3.0, May 2008. http://www.openmp.org/mp-documents/spec30.pdf.

[26] Raghavan Raman, Jisheng Zhao, Zoran Budmilic, and Vivek Sarkar. Compiler support for work-stealing parallel runtime systems. Technical Report TR10-1, Department of Computer Science, Rice University, January 2010.

[27] Raghavan Raman, Jisheng Zhao, Vivek Sarkar, Martin T. Vechev, and Eran Yahav. Efficient data race detection for async-finish parallelism. In *RV*, pages 368–383, 2010.

[28] Vivek Sarkar. A Concurrent Execution Semantics for Parallel Program Graphs and Program Dependence Graphs (Extended Abstract). *Springer-Verlag Lecture Notes in Computer Science*, 757:16–30, 1992. LCPC'92.

[29] Vivek Sarkar. Automatic Selection of High Order Transformations in the IBM XL Fortran Compilers. *IBM Journal of Research and Development*, 41(3), May 1997.

[30] Vivek Sarkar. Analysis and Optimization of Explicitly Parallel Programs using the Parallel Program Graph Representation. In *LCPC'98*, Lecture Notes in Computer Science. Springer-Verlag, New York, 1998.

[31] Vivek Sarkar and Barbara Simons. Parallel Program Graphs and their Classification. *Springer-Verlag Lecture Notes in Computer Science*, 768:633–655, 1993. LCPC'93.

[32] J. Shirako et al. Phasers: a unified deadlock-free construct for collective and point-to-point synchronization. In *ICS '08*, pages 277–288, New York, NY, USA, 2008. ACM.

[33] Jun Shirako, Jisheng Zhao, V. Krishna Nandivada, and Vivek Sarkar. Chunking parallel loops in the presence of synchronization. In *ICS'09*, pages 181–192, 2009.

[34] R. Vallée-Rai et al. Soot - a Java Optimization Framework. In *Proceedings of CASCON 1999*, pages 125–135, 1999.

[35] T.J. Watson Libraries for Analysis (WALA). http://wala.sourceforge.net/wiki/index.php/MainPage, Apr 2010.

[36] Edwin Westbrook, Jisheng Zhao, Zoran Budimlic, and Vivek Sarkar. Permission Regions for Race-Free Parallelism. In *Proceedings of RV 2011*, 2011.

[37] X10 release on SourceForge. http://x10.sf.net.

[38] Yonghong Yan et al. Hierarchical place trees: A portable abstraction for task parallelism and data movement. In *LCPC 2009*. Springer, 2009.

[39] K. Yelick et al. Productivity and performance using partitioned global address space languages. In *Proceedings of the international workshop on Parallel symbolic computation*, pages 24–32, New York, NY, USA, 2007. ACM.

[40] Jisheng Zhao and others Shirako. Reducing task creation and termination overhead in explicitly parallel programs. In *PACT'10*, New York, NY, USA, 2010. IEEE Computer Society.