SPIRE: A Methodology for Sequential to Parallel Intermediate Representation Extension

Dounia Khaldi, Pierre Jouvelot, François Irigoin and Corinne Ancourt

CRI, Mathématiques et systèmes, MINES ParisTech
35 rue Saint-Honoré, 77300 Fontainebleau, France
`firstname.lastname@mines-paristech.fr`

Abstract. SPIRE is a new methodology for the design of parallel extensions of the intermediate representations used in compilation frameworks of sequential languages. It can be used to leverage existing infrastructures for sequential languages to address both control and data parallel constructs while preserving as much as possible existing analyses for sequential and parallel code. We suggest to view this upgrade process as an "intermediate representation transformer" at the syntactic and semantic levels; we show this can be done via the introduction of only ten new concepts, collected in three groups, namely execution, synchronization and data distribution, and precisely defined via a formal semantics and rewriting rules.
We use the sequential intermediate representation of PIPS, a comprehensive source-to-source compilation platform, as a use case for our approach. We introduce our SPIRE parallel primitives, extend PIPS intermediate representation and show how example code snippets from the OpenCL, Cilk, OpenMP, X10, Habanero-Java, MPI and Chapel parallel programming languages can be represented this way. A formal definition of SPIRE operational semantics is provided, built on top of the one used for the sequential intermediate representation. We assess the generality of our proposal by (1) showing how a different sequential IR, namely LLVM, can be extended to handle parallelism using the SPIRE methodology and (2) providing implementation and run-time performance data of a SPIRE-derived parallelization process.
Our primary goal with the development of SPIRE is to provide, at a low cost, powerful parallel program representations that will ease the design of efficient automatic parallelization algorithms. More generally, our work provides a possible roadmap for the compiler designers who need to introduce parallel features into their own infrastructures.

Keywords: parallel intermediate representation, operational semantics, PIPS, LLVM

1 Introduction

The growing importance of parallel computers and the search for an efficient programming model led, and is still leading, to a proliferation of parallel programming languages such as, currently, Cilk [1], Chapel [2], X10 [3], Habanero-Java [4], OpenMP [5], OpenCL [6] or MPI [7]. To adapt to such an evolution, compilers need to introduce internal intermediate representations (IR) for parallel programs. The choice of a proper parallel IR is of key importance, since the efficiency and power of the transformations and optimizations these compilers can perform are closely related to the selection of a proper program representation paradigm. Yet, given the wide variety of the existing programming models, it would be better, from a software engineering point of view, to find a unique parallel IR, as general and simple as possible.

Existing proposals for program representation techniques already provide a basis for the exploitation of parallelism via the encoding of control and/or data flow information. HPIR [8], PLASMA [9] or InsPIRe [10] are instances that operate at a high abstraction level, while the hierarchical task, stream or program dependence graphs (we survey these notions in Section 2) are better suited to graph-based approaches. Yet many more existing compiler frameworks use traditional representations for sequential-only programs, and changing their internal data structures or adding ad-hoc built-ins to deal with parallel constructs is a difficult and time-consuming task.

The main motivation behind the design of the methodology introduced in our paper is to preserve the many years of development efforts invested in huge compiler platforms such as GCC (more than 7 million lines of code), PIPS (600 000 lines of code), LLVM (more than 1 million lines of code),... when upgrading their intermediate representations to handle parallel languages, as source languages or as targets for source-to-source transformations. We provide an evolutionary path for these large software developments via the introduction of the Sequential to Parallel Intermediate Representation Extension (SPIRE) methodology that we show that can be plugged into existing compiler projects in a rather simple manner.

SPIRE is based on only three key concepts: (1) the parallel vs. sequential execution of groups of statements such as sequences, loops and general control-flow graphs, (2) the global synchronization characteristics of statements and the specification of finer grain synchronization via the notion of events and (3) the handling of data distribution for different memory models. To illustrate how this approach can be used in practice, we use SPIRE to extend the intermediate representation (IR) [11] of PIPS [12], a comprehensive source-to-source compilation and optimization platform, and of LLVM [13], a widespread SSA-based compilation infrastructure.

The design of SPIRE is the result of many trade-offs between generality and precision, abstraction and low-level concerns. On the one hand, and in particular when looking at source-to-source optimizing compiler platforms adapted to multiple source languages, one needs to be able to represent as many of the existing (and, hopefully, future) parallel constructs while minimizing the number of new concepts introduced in the parallel IR. Yet, keeping only a limited number of hardware-level notions in the IR, while good enough to deal with all parallel constructs, would entail convoluted rewritings of high-level parallel flows. We used an extensive survey of key parallel languages, namely Cilk, Chapel, X10, Habanero-Java, OpenMP, OpenCL and MPI, to guide our design of SPIRE, while showing how to express their relevant parallel constructs within SPIRE.

The four contributions of this paper are:

- SPIRE, a new, simple, parallel intermediate representation extension methodology for designing the parallel IRs used in compilation frameworks; it leverages their existing infrastructure for sequential languages to address control and data parallelism and data distribution;
- the application of SPIRE to the IR used in the PIPS compilation framework, yielding a parallel IR to be used for both automatic task-level parallelization and the optimization of explicitly parallel programs;
- the small-step, operational semantics of the SPIRE transformation process, to formally define how its key parallel concepts are added to existing systems;
- an evaluation of the generality of SPIRE, by showing how our methodology can be applied to another IR, namely the IR of the widely-used LLVM system, and providing implementation and run-time performance data for a SPIRE-derived parallelization process.

After this introduction, we survey existing parallel IRs in Section 2. We describe our use-case sequential IR, used by the PIPS compilation framework, in Section 3. Our parallel extension proposal, SPIRE, is introduced in Section 4, where we also show how simple illustrative examples written in OpenCL, Cilk, OpenMP, X10, Habanero-Java, MPI and Chapel can be easily represented within SPIRE. The formal operational semantics of SPIRE is given in Section 5. Section 6 shows the generality of SPIRE by showing its use on LLVM and discusses performance results. We discuss future work and conclude in Section 7.

2 Related Work

In this section, we review several different possible representations of parallel programs, both at the high, syntactic, and mid, graph-based, levels. We provide synthetic descriptions of the key existing IRs addressing similar issues to our paper's. Bird's eye view comparisons with SPIRE are also given here, although a more extensive analysis of existing IRs would require more space than permitted by the paper format.

Syntactic approaches to parallelism expression use abstract syntax tree nodes, while adding specific parallel built-in functions. For instance, the intermediate representation of the implementation of OpenMP in GCC (GOMP) [14] extends its three-address representation, GIMPLE [15]. The OpenMP parallel directives are replaced by specific built-ins in low- and high-level GIMPLE, and additional nodes in high-level GIMPLE, such as the `__sync_fetch_and_add` built-in function for an atomic memory access addition. Similarly, Sarkar and Zhao introduce the high-level parallel intermediate representation HPIR [8] that decomposes Habanero-Java programs into region syntax trees, while maintaining additional data structures on the side: region control-flow graphs and region dictionaries. New program instructions are introduced: `AsyncRegionEntry` and `AsyncRegionExit` delimit tasks, while `FinishRegionEntry` and `FinishRegionExit` can be used in parallel sections. SPIRE borrows some of the ideas used in GOMP or HPIR, but frames them in more structured settings while trying to be more language-neutral; in particular, we try to minimize the number of additional built-in functions, which have the drawback of hiding the abstract high-level structure of parallelism. Applying the SPIRE approach to systems such as GCC would have provided a minimal set of extensions that could have also been used for other implementations of parallel languages that rely on GCC as a backend, such as Cilk.

PLASMA is a programming framework for heterogeneous SIMD systems, with an IR [9] that abstracts data parallelism and vector instructions. It provides specific operators such as `add` on vectors and special instructions such as `reduce` and `par`. While PLASMA abstracts SIMD implementation and compilation concepts for SIMD accelerators, SPIRE is more architecture-independent and also covers control parallelism.

InsPIRe is the parallel intermediate representation at the core of the source-to-source Insieme compiler [10] for C, C++, OpenMP, MPI and OpenCL programs. Parallel constructs are encoded using built-ins. SPIRE intends to also cover source-to-source optimization. We believe it could have been applied to Insieme sequential components, parallel constructs being defined as extensions of the sequential abstract syntax tree nodes of InsPIRe instead of using numerous built-ins.

Turning now to mid-level intermediate representations, many systems rely on graph structures for representing sequential code, and extend them for parallelism. The Hierarchical Task Graph [16] represents the program control flow. The hierarchy exposes the loop nesting structure; at each loop nesting level, the loop body is hierarchically represented as a single node that embeds a subgraph that has control and data dependence information associated with it. SPIRE is able to represent both structured and unstructured control-flow dependence, thus enabling recursively-defined optimization techniques to be applied easily. The hierarchical nature of underlying sequential IRs can be leveraged, via SPIRE, to their parallel extensions; this feature is used in the PIPS case addressed below.

A stream graph [17] is a dataflow representation introduced specifically for streaming languages. Nodes represent data reorganization and processing operations between streams, and edges, communications between nodes. The number of data samples defined and used by each node is supposed to be known statically. Each time a node is fired, it consumes a fixed number of elements of its inputs and produces a fixed number of elements on its outputs. SPIRE provides support for both data and control dependence information; streaming can be handled in SPIRE using its point-to-point synchronization primitives.

The parallel program graph (PPDG) [18] extends the program dependence graph [19], where vertices represent blocks of statements and edges, essential control or data dependences; *mgoto* control edges are added to represent task creation occurrences, and synchronization edges, to impose ordering on tasks. Kimble IR [20] uses an intermediate representation designed along the same lines, i.e., as a hierarchical direct acyclic graphs (DAG) on top of GCC IR, GIMPLE. Parallelism is expressed there using new types of nodes: *region*, which is a subgraph of *cluster*s, and *cluster*, a sequence of statements to be executed by one thread. Like PPDG and Kimble IR, SPIRE adopts an extension approach to "parallelize" existing sequential intermediate representations; our paper shows that this can be defined as a general mechanism for parallel IR definitions and provides a formal specification of this concept.

LLVM IR [13] represents each function as control flow graphs. To encode parallel constructs, LLVM introduces the notion of metadata such as `llvm.loop.parallel` for implementing parallel loops. A metadata is a string used as an annotation on the LLVM IR nodes. LLVM IR lacks support for other parallel constructs such as starting parallel threads, synchronizing them, etc. We present in Section 6.1 our proposal for a parallel IR for LLVM via the application of SPIRE to LLVM.

3 PIPS (Sequential) IR

Since this paper introduces SPIRE as an *extension* formalism for existing intermediate representations, a sequential, base case IR is needed to present our proposal. We chose the IR of PIPS [12] to showcase our approach, since it is readily available, well-documented and encodes both control and data dependences. To help support our claim of the generality of our approach, Section 6.1 illustrates another application of SPIRE, this time on LLVM [13].

PIPS is a powerful source-to-source compilation and optimization platform; its intermediate representation (IR) [11] of sequential programs is a hierarchical data structure that embeds both control flow graphs and abstract syntax trees. To describe SPIRE, we show how to extend this IR to parallel programs in order to obtain an abstraction for parallel languages for optimization and transformation purposes.

We provide in this section a high-level description of the intermediate representation of PIPS; it is specified using Newgen [21], a Domain Specific Language for the definition of set equations from which a dedicated API is automatically generated to manipulate (creation, access, IO operations...) data structures implementing these set elements. Since our purpose is to highlight the design of parallel extensions, many of these sets remain unchanged; this section contains only a slightly simplified subset of the intermediate representation of PIPS, the part that is directly related to the parallel paradigms in SPIRE. The Newgen definition of this part is given in the Figure 1:

- Control flow in PIPS IR is represented via instructions, members of the disjoint union (using the "+" symbol) set `instruction`. An instruction can be either a simple call or a compound instruction, i.e., a `for` loop, a sequence or a control flow graph. A call instruction represents built-in or user-defined function calls; for instance, assign statements are represented as calls to the "`:=`" function. The `call` set is not defined here.
- Instructions are included within statements, which are members of a Cartesian product set that also incorporates the declarations of local variables; thus a whole function is represented in PIPS IR as a statement. In Newgen, a given set component can be distinguished using a prefix such as `declarations`

here; all named objects such as user variables or built-in functions in PIPS are members of the `entity` set (the `value` set denotes constants while the "*" symbol introduces Newgen list sets).

– Compound instructions can be either (1) a loop instruction, which includes an iteration index variable with its lower, upper and increment expressions and a loop body (the `expression` set definition is not provided here), (2) a sequence, i.e., a succession of statements, encoded as a list, or (3) an unstructured control flow graph.
– Programs that contain structured (`continue`, `break` and `return`) and unstructured (`goto`) transfers of control are handled in the PIPS intermediate representation via the `unstructured` set. An unstructured instruction has one entry and one exit `control` node; a control is a node in a graph labeled with a statement and its lists of predecessor and successor control nodes. Executing an unstructured instruction amounts to following the control flow induced by the graph successor relationship, starting at the entry node, while executing the node statements, until the exit node is reached.

```
instruction  = call + forloop + sequence + unstructured;
statement    = instruction x declarations:entity*;
entity       = name:string x type x initial:value;
forloop      = index:entity x lower:expression x upper:expression x
               step:expression x body:statement;
sequence     = statements:statement*;
unstructured = entry:control x exit:control;
control      = statement x predecessors:control* x successors:control*;
```

Fig. 1: Simplified Newgen definitions of the PIPS IR

4 SPIRE, a Sequential to Parallel IR Extension Methodology

In this section, we present in detail the SPIRE methodology, which can be used to add parallel concepts to sequential IRs. After introducing our design philosophy, we describe the application of SPIRE on the PIPS IR. We illustrate these SPIRE-derived constructs with code excerpts from various parallel programming languages; our intent is not to provide here general rewriting techniques from these to SPIRE (this would be way out of the scope of this paper), but to provide hints on how such rewritings might possibly proceed. Note that, in Section 6.1, using LLVM, we show that our approach is general enough to be adapted to other IRs.

4.1 Design Approach

SPIRE intends to be a practical methodology to extend existing sequential IRs to adapt to parallelism issues, either to generate parallel code from sequential programs or address explicitly parallel programming languages. Interestingly, the idea of seeing the issue of parallelism as an extension over sequential concepts is in sync with Dijkstra's view that "parallelism or concurrency are operational concepts that refer not to the program, but to its execution." [22]. If one accepts such a vision, adding parallelism extensions to existing IRs, as advocated by our approach with SPIRE, can thus, at a fundamental level, not be seen as an afterthought but as a consequence of the fundamental nature of parallelism.

Our design of SPIRE does not intend to be minimalist but to be as seamlessly as possible integrable within actual IRs, while able to handle as many parallel programming constructs as possible. To be successful, our design point must provide proper trade-offs between generality, expressibility and conciseness of representation. We used an extensive survey of existing parallel languages to guide us during this design process. Table 1, which extends the one provided in [23], summarizes the main characteristics of seven recent and widely used parallel languages: Cilk, Chapel, X10, Habanero-Java, OpenMP, OpenCL and MPI. The main constructs used in each language to launch task and data parallel computations, perform synchronization, introduce atomic sections and transfer data in the various memory models are listed. Our main finding from this analysis is that, to be able to deal with parallel programming, one simply needs to add to a given sequential IR the ability to specify (1) the parallel execution mechanism of groups of statements, (2) the synchronization behavior of statements and (3) the layout of data, i.e., how memory is modeled in the parallel language.

The last line of Table 1 summarizes the approach we propose to map these programming concepts to our parallel intermediate representation extension. SPIRE is based on the introduction of only ten key notions, collected in three groups:

– execution, via the `sequential` and `parallel` constructs;
– synchronization, via the `spawn`, `barrier`, `atomic`, `single`, `signal` and `wait` constructs;
– data distribution, via `send` and `recv` constructs.

Small code snippets are provided below to sketch how the key constructs of these parallel languages can be encoded in practice within a SPIRE-extended parallel IR.

4.2 Execution

The issue of parallel vs. sequential execution appears when dealing with groups of statements, which in our case study correspond to members of the `forloop`, `sequence` and `unstructured` sets. To apply SPIRE to PIPS sequential IR, an `execution` attribute is added to these sequential set definitions:

```
forloop'      = forloop x execution;
sequence'     = sequence x execution;
unstructured' = unstructured x execution;
```

The primed sets `forloop'` (expressing data parallelism) and `sequence'` and `unstructured'` (implementing control parallelism) represent SPIREd-up sets for the PIPS parallel IR. Of course, the ‘prime’ notation is used here for pedagogical purpose only; in practice, an execution field is added in the existing IR representation. The definition of `execution` is straightforward:

```
execution = sequential:unit + parallel:unit;
```

where `unit` denotes a set with one single element; this encodes a simple enumeration of cases for execution. A `parallel` execution attribute asks for all loop iterations, sequence statements and control nodes of unstructured instructions to be run concurrently, using an implicit fork/join metaphor.

For instance, a parallel `execution` construct can be used to represent data parallelism on GPUs, when expressed via the OpenCL `clEnqueueNDRangeKernel` function (see Figure 2). This function call could be encoded within PIPS parallel IR as a parallel loop, each iteration executing the `kernel`

Language	Execution	Synchronization				Memory	
	Parallelism	Task creation	Task join	Point-to-point	Atomic section	Model	Data distribution
Cilk (MIT)	—	spawn	sync	—	cilk_lock	*Shared*	—
Chapel (Cray)	forall coforall cobegin	begin	—	sync	sync atomic	*PGAS (Locales)*	(on)
X10 (IBM)	foreach	async future	finish	next force	atomic	*PGAS (Places)*	(at)
Habanero-Java (Rice)	foreach	async future	finish	next get	atomic isolated	*PGAS (Places)*	(at)
OpenMP	omp for omp sections	omp task omp section	omp taskwait omp barrier	—	omp critical omp atomic	*Shared*	private, shared...
OpenCL	EnqueueND-RangeKernel	EnqueueTask	Finish EnqueueBarrier	*events*	atom_add, ...	*Distributed*	ReadBuffer WriteBuffer
MPI	MPI_Init	MPI_spawn	MPI_Finalize MPI_Barrier	—	—	*Distributed*	MPI_Send MPI_Recv...
SPIRE	sequential, parallel	spawn	barrier	signal, wait	atomic	*Shared, Distributed*	send, recv

Table 1. Mapping of SPIRE to parallel languages constructs (terms in parentheses are not currently handled by SPIRE)

```
//Execute 'n' kernels in parallel
global_work_size[0] = n;
err = clEnqueueNDRangeKernel(cmd_queue, kernel, 1, NULL,
                   global_work_size, NULL, 0, NULL, NULL);
```

Fig. 2: OpenCL example illustrating a parallel loop

function as a separate task, receiving the proper index value as an argument. In this paper, a task is a statement, to be executed by a thread.

An another example, in the left side of Figure 3, from Chapel, illustrates its `forall` data parallelism construct, which will be encoded with a SPIRE parallel loop.

```
forall I in 1..n do
    t[i] = 0;
```

```
forloop(I,1,n,1,
        t[i] = 0,
        parallel)
```

Fig. 3: `forall` in Chapel, and its SPIRE core language representation

4.3 Synchronization

The issue of synchronization is a characteristic feature of the run-time behavior of one statement with respect to other statements. SPIRE extends sequential intermediate representations in a straightforward way by adding a synchronization attribute to the specification of statements:

```
statement' = statement x synchronization;
```

Coordination by synchronization in parallel programs is often dealt via coding patterns such as barriers, used for instance when a code fragment contains many phases of parallel execution where each phase should wait for the precedent ones to proceed. We define the `synchronization` set via high-level coordination characteristics useful for optimization purposes:

```
synchronization = none:unit + spawn:entity + barrier:unit +
                  single:bool + atomic:reference;
```

When S is the statement with the synchronization attribute:

- `none` specifies the default behavior, i.e., independent with respect to other statements, for S;
- `spawn` induces the creation of an asynchronous task S, while the value of the corresponding entity is the user-chosen number of the thread that executes S;
- `barrier` specifies that all the child threads spawned by the execution of S are suspended before exiting until they are all finished – an OpenCL example illustrating spawn (`clEnqueueTask`) and barrier (`clEnqueueBarrier`) is provided in Figure 4;

```
mode = OUT_OF_ORDER_EXEC_MODE_ENABLE;
commands = clCreateCommandQueue(context, device_id,mode,&err);
clEnqueueTask(commands, kernel_A, 0, NULL, NULL);
clEnqueueTask(commands, kernel_B, 0, NULL, NULL);
// synchronize so that Kernel C starts only after Kernels A and B have finished
clEnqueueBarrier(commands);
clEnqueueTask(commands, kernel_C, 0, NULL, NULL);
```

Fig. 4: OpenCL example illustrating spawn and barrier statements

- `single` ensures that S is executed by only one thread in its thread team (a thread team is the set of all the threads spawned within the innermost parallel `forloop` statement) and a barrier exists at the end of a single operation if its `synchronization_single` value is true;
- `atomic` predicates the execution of S to the acquisition of a lock to ensure exclusive access; at any given time, S can be executed by only one thread. Locks are logical memory addresses, represented here by a member of the PIPS IR `reference` set (not specified in this paper). An example illustrating how an atomic synchronization on the reference `l` in a statement modifying Array `x` can be translated in Cilk (via `Cilk_lock` and `Cilk_unlock`) and OpenMP (`critical`) is provided in Figure 5.

```
Cilk_lockvar l;                          
Cilk_lock_init(l);
...                                      #pragma omp critical
Cilk_lock(l);                              x[index[i]] += f(i);
 x[index[i]] += f(i);
Cilk_unlock(l);
```

Fig. 5: Cilk and OpenMP examples illustrating an atomically-synchronized statement

4.4 Event API

In parallel code, one usually distinguishes between two types of synchronization: (1) coarse grain (collective) synchronization between threads using barriers, which are handled in our SPIRE methodology via the `synchronization` patterns defined above, and (2) fine grain (point-to-point) synchronization between participating threads. Handling point-to-point synchronization using extensions on abstract syntax tree nodes, as done up to now, is too constraining when one has to deal with a varying set of threads that may belong to different parallel parent nodes. Thus, SPIRE suggests to deal with this last class of coordination using a new class of values, of the `event` type.

SPIRE extends the `type` set of entities with a new basic type, namely `event`:

```
type′ = type + event:unit ;
```

Values of type `event` are counters, in a manner reminiscent of semaphores [24]. The programming interface for events is defined by the following functions:

- `event newEvent(int i)` is the creation function of events, initialized with the integer `i` that specifies how many threads can execute `wait` on this event without being blocked;
- `void signal(event e)` increments by one the event value of `e`. Note that The `void` return type will be replaced by `int` in practice, to enable the handling of error values;
- `void wait(event e)` blocks the thread that calls it until the value of `e` is strictly greater than 0. When the thread is released, this value is decremented by one.

In a first example of possible use of this event API, the construct `future` used in X10 (see Figure 6) can be seen as the spawning of the computation of `foo()`. The end result is obtained via the call to the `force` method; such a mechanism can be easily implemented in SPIRE using an event attached to the running task; it is `signal`ed when the task is completed and `wait`ed by the `force` method.

```
future<int> Fi = future{foo()};
int i = Fi.force();
```

Fig. 6: X10 example illustrating a future task and its synchronization

A second example, taken from Habanero-Java, illustrates how point-to-point synchronization primitives such as phasers and the `next` statement can be dealt with using the Event API (see Figure 7, left). The `async phased` keyword can be replaced by `spawn`. In this example, the `next` statement is equivalent to the following sequence:

```
signal(ph);
wait(ph);
signal(ph);
```

where the event ph is supposed initialized to newEvent (-(n-1)); the second signal is used to resume the suspended tasks in a chain-like fashion.

```
finish{
 phaser ph = new phaser();
 for(j = 1;j <= n;j++){
  async phased(
         ph<SIG_WAIT>){
   S;
   next;
   S′;
  }
 }
}
```

```
barrier(
 ph = newEvent(-(n-1));
 j = 1;
 loop(j <= n,
  spawn(j,
        S;
        signal(ph);
        wait(ph);
        signal(ph);
        S');
  j = j+1))
```

Fig. 7: A phaser in Habanero-Java, and its SPIRE core language representation

4.5 Data Distribution

The choice of a proper memory model to express parallel programs is an important issue when designing a generic intermediate representation. There are usually two main approaches to memory modeling: shared and message passing models. Since SPIRE is designed to extend existing IR for sequential languages, it can be straightforwardly seen as using a shared memory model when parallel constructs are added. By convention, we say that spawn creates processes, in the case of message passing memory models, and threads, in the other case.

In order to take into account the explicit distribution required by the message passing memory model used in parallel languages such as MPI, SPIRE introduces the send and recv blocking functions for implementing communication between processes:

– void send(int dest, entity buf) transfers the value in Entity buf to the process numbered dest;
– void recv(int source, entity buf) receives in buf the value sent by Process source.

Note that non-blocking communications can be easily implemented in SPIRE using the above primitives within spawned statements. Also, broadcast collective communications, such as defined in MPI, can be seen as wrappers around send and recv operations. When the master process and receiver processes want to perform a broadcast function, then, if this process is the master, its broadcast operation is equivalent to a loop over receivers, with a call to send as body; otherwise (receiver), the broadcast is a recv function.

The MPI example in Figure 8 can be represented in SPIRE as a sequential loop with index my_rank of size iterations whose body spawns the MPI code from MPI_Comm_size to MPI_Finalize, using

```
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);
if (my_rank == 0)
  MPI_Recv(sum,sizeof(sum),MPI_FLOAT,1,1,MPI_COMM_WORLD,&stat);
else if(my_rank == 1){
  sum = 42;
  MPI_Send(sum,sizeof(sum),MPI_FLOAT,0,1,MPI_COMM_WORLD);
}
MPI_Finalize();
```

Fig. 8: MPI example illustrating a communication

`my_rank` as process number. The communication of Variable `sum` from Process 1 to Process 0 can be handled with SPIRE `send`/`recv` functions.

An other interesting memory model for parallel programming has been introduced somewhat recently: the Partitioned Global Address Space [25]. The uses of the PGAS memory model in languages such as UPC [26], Habanero-Java, X10 and Chapel introduce various notions such as `Place` or `Locale` to label portions of a logically-shared memory that threads may access, in addition to complex APIs for distributing data over these portions. Given the wide variety of current proposals, we leave the issue of integrating the PGAS model within the general methodology of SPIRE as future work.

5 SPIRE Operational Semantics

The purpose of the formal definition described in this section is to provide a solid basis for program analyses and transformations. It is a systematic way to specify our IR extension mechanism, something seldom present in IR definitions. It also illustrates how SPIRE leverages the syntactic and semantic level of sequential constructs to parallel ones, preserving the sequential traits and, thus, analyses.

Fundamentally, at the syntactic and semantic levels, SPIRE is a methodology for expressing representation transformers, mapping the definition of a sequential language IR to a parallel version. We define the operational semantics of SPIRE in a two-step fashion: we introduce (1) a minimal core parallel language that we use to model fundamental SPIRE concepts and for which we provide a small-step operational semantics and (2) rewriting rules that translate the more complex constructs of SPIRE in this core language.

5.1 Sequential Core Language

Illustrating the transformations induced by SPIRE requires the definition of a sequential IR basis, as was done above, via PIPS IR. Since we focus here on the fundamentals, we use as core language a simpler, minimal sequential language, `Stmt`. Its syntax is given in Figure 9, where we assume that the sets `Ide` of identifiers `I` and `Exp` of expressions `E` are given.

Sequential statements are: (1) `nop` for no operation, (2) `I=E` for an assignment of `E` to `I`, (3) $\texttt{S}_1\texttt{;S}_2$ for a sequence and (4) `loop(E,S)` for a while loop.

At the semantic level, a statement in `Stmt` is a very simple memory transformer. A memory $m \in Memory$ is a mapping in $\texttt{Ide} \rightarrow Value$, where values $v \in Value = N + Bool$ can either be integers

```
S ∈ Stmt::=
    nop | I=E | S₁;S₂ | loop(E,S)

S ∈ SPIRE(Stmt)::=
    nop | I=E | S₁;S₂ | loop(E,S) |
    spawn(I,S) |
    barrier(S) | barrier_wait(n) |
    wait(I) | signal(I) |
    send(I,I′) | recv(I,I′)
```

Fig. 9: `Stmt` and `SPIRE(Stmt)` syntaxes

$n \in N$ or booleans $b \in Bool$. The sequential operational semantics for `Stmt`, expressed as transition rules over configurations $\kappa \in Configuration = Memory \times \texttt{Stmt}$, is given in Figure 10; we assume that the program is syntax- and type-correct. A transition $(m, \texttt{S}) \rightarrow (m', \texttt{S}')$ means that executing the statement S in a memory m yields a new memory m' and a new statement `S'`; we posit that the "$\rightarrow$" relation is transitive. Rules 1 to 5 encode typical sequential small-step operational semantic rules for the sequential part of the core language. We assume that $\zeta \in \texttt{Exp} \rightarrow Memory \rightarrow Value$ is the usual function for expression evaluation.

$$\frac{v = \zeta(\texttt{E})m}{(m, \texttt{I} = \texttt{E}) \rightarrow (m[\texttt{I} \rightarrow v], \texttt{nop})} \tag{1}$$

$$(m, \texttt{nop}; \texttt{S}) \rightarrow (m, \texttt{S}) \tag{2}$$

$$\frac{(m, \texttt{S}_1) \rightarrow (m', \texttt{S}_1')}{(m, \texttt{S}_1; \texttt{S}_2) \rightarrow (m', \texttt{S}_1'; \texttt{S}_2)} \tag{3}$$

$$\frac{\zeta(\texttt{E})m}{(m, \texttt{loop(E,S)}) \rightarrow (m, \texttt{S}; \texttt{loop(E,S)})} \tag{4}$$

$$\frac{\neg\zeta(\texttt{E})m}{(m, \texttt{loop(E,S)}) \rightarrow (m, \texttt{nop})} \tag{5}$$

Fig. 10: `Stmt` sequential transition rules

The semantics of a whole sequential program `S` is defined as the memory m such that $(\perp, \texttt{S}) \rightarrow (m, \texttt{nop})$, if the execution of `S` terminates.

5.2 SPIRE as a Language Transformer

Syntax At the syntactic level, SPIRE specifies how a grammar for a sequential language such as `Stmt` is transformed, i.e., extended, with synchronized parallel statements. The grammar of `SPIRE(Stmt)` in

Figure 9 adds to the sequential statements of `Stmt` (from now on, synchronized using the default `none`) new parallel statements: a task creation `spawn`, a termination `barrier` and two `wait` and `signal` operations on events or `send` and `recv` operations for communication. Synchronizations `single` and `atomic` are defined via rewriting (see Subsection 5.3). The statement `barrier_wait(`n`)`, added here for specifying the multiple-step behavior of the `barrier` statement in the semantics, is not accessible to the programmer. Figure 7 provides the SPIRE representation of a program example.

Semantic domains As SPIRE extends grammars, it also extends semantics. The set of values manipulated by `SPIRE(Stmt)` statements extends the sequential $Value$ domain with events $e \in \textit{Event} = N$, that encode events current values; we posit that $\zeta(\texttt{newEvent(E)})m = \zeta(\texttt{E})m$.

Parallelism is managed in SPIRE via processes (or threads). We introduce control state functions $\pi \in State = Proc \rightarrow Configuration \times Procs$ to keep track of the whole computation, mapping each process $i \in Proc = N$ to its current configuration (i.e., the statement it executes and its own view of memory) and the set $c \in Procs = \wp(Proc)$ of the process children it has spawned during its execution.

In the following, we note $dom(\pi) = \{i \in Proc/\pi(i)$ is defined$\}$ the set of currently running processes, and $\pi[i \rightarrow (\kappa, c)]$ the state π extended at i with (κ, c). A process is said to be *finished* if and only if all its children processes, in c, are also finished, i.e., when only `nop` is left to execute: $finished(\pi, c) = (\forall i \in c, \exists c_i \in Procs, \exists m_i \in Memory/\pi(i) = ((m_i, \texttt{nop}), c_i) \wedge finished(\pi, c_i))$.

Memory Models The memory model of sequential languages is a unique address space for identifier values. In our parallel extension, a configuration for a given process or thread includes its view of memory. We suggest to use the same semantic rules, detailed below, to deal with both shared and message passing memory rules. The distinction between these models, beside the additional use of send/receive constructs in the message passing model versus events in the shared one, is included in SPIRE via constraints we impose on the control states π used in computations. Namely, we posit that, in the shared memory model, for all threads i and i' with $\pi(i) = ((m, \texttt{S}), c)$ and $\pi(i') = ((m', \texttt{S}'), c')$, one has $m = m'$. No such constraint is needed for the message passing model. Regarding the notion of memory equality, note that the issue of private variables in threads would have to be introduced in full-fledged languages. As mentioned above, PGAS is left for future work; some sort of constraints based on the characteristics of the address space partitioning for places/locales would have to be introduced.

Semantic Rules At the semantic level, SPIRE is thus a transition system transformer, mapping rules such as the ones in Figure 10 to parallel, synchronized transition rules in Figure 11. A transition $(\pi[i \rightarrow ((m, \texttt{S}), c)]) \hookrightarrow (\pi'[i \rightarrow ((m', \texttt{S}'), c')])$ means that the i-th process, when executing S in a memory m, yields a new memory m' and a new control state $\pi'[i \rightarrow ((m', \texttt{S}'), c')]$ in which this process now will execute S'; additional children processes may have been created in c' compared to c. We posit that the "$\hookrightarrow$" relation is transitive.

Rule 6 is a key rule to specify SPIRE transformer behavior, providing a bridge between the sequential and the SPIRE-extended parallel semantics; all processes can non-deterministically proceed along their sequential semantics "$\rightarrow$", leading to valid evaluation steps along the parallel semantics "$\hookrightarrow$". The interleaving between parallel processes in `SPIRE(Stmt)` is a consequence of (1) the non-deterministic choice of the value of i within $dom(\pi)$ when selecting the transition to perform and (2) the number of steps executed by the sequential semantics. Note that one might want to add even more non-determinism in our semantics; indeed, Rule 1 is atomic: loading the variables in `E` and performing the store operation to `I` are performed in one sequential step. To simplify this presentation, we do not provide the simple intermediate steps in the sequential evaluation semantics of Rule 1 that would have removed this artificial atomicity.

The remaining rules focus on parallel evaluation. In Rule 7, `spawn` adds to the state a new process n that executes `S` while inheriting the parent memory m in a fork-like manner; the set of processes spawned by n is initially equal to ϕ, and n is added to the set of processes c spawned by i. Rule 8 implements a rendezvous: a new process n executes S, while process i is suspended as long as $finished$ is not true; indeed, the rule 9 resumes execution of process i when all the child processes spawned by n have finished. In Rules 10 and 11, `I` is an `event`, that is a counting variable used to control access to a resource or to perform a point-to-point synchronization, initialized via `newEvent` to a value equal to the number of processes that will be granted access to it. Its current value n is decremented every time a `wait(I)` statement is executed and, when $\pi(\texttt{I}) = n$ with $n > 0$, the resource can be used or the barrier can be crossed. In Rule 11, the current value n' of `I` is incremented; this is a non-blocking operation. In Rule 12, p and p' are two processes that communicate: p sends the datum `I` to p', while this later consumes it in `I'`.

$$\frac{\kappa \rightarrow \kappa'}{\pi[i \rightarrow (\kappa, c)] \hookrightarrow \pi[i \rightarrow (\kappa', c)]} \tag{6}$$

$$\frac{n = \zeta(\texttt{I})m}{\pi[i \rightarrow ((m, \texttt{spawn(I,S)}), c)] \hookrightarrow \pi[i \rightarrow ((m, \texttt{nop}), c \cup \{n\})][n \rightarrow ((m, \texttt{S}), \emptyset)]} \tag{7}$$

$$\frac{n \notin dom(\pi) \cup \{i\}}{\pi[i \rightarrow ((m, \texttt{barrier(S)}), c)] \hookrightarrow \pi[i \rightarrow (m, \texttt{barrier\_wait}(n)), c)][n \rightarrow ((m, \texttt{S}), \emptyset)]} \tag{8}$$

$$\frac{finished(\pi, \{n\}) \wedge \pi(n) = ((m', \texttt{nop}), c')}{\pi[i \rightarrow ((m, \texttt{barrier\_wait}(n)), c)] \hookrightarrow \pi[i \rightarrow ((m', \texttt{nop}), c)]} \tag{9}$$

$$\frac{(n = \zeta(\texttt{I})m) \wedge (n > 0)}{\pi[i \rightarrow ((m, \texttt{wait(I)}), c)] \hookrightarrow \pi[i \rightarrow ((m[\texttt{I} \rightarrow n - 1], \texttt{nop}), c)]} \tag{10}$$

$$\frac{n = \zeta(\texttt{I})m}{\pi[i \rightarrow ((m, \texttt{signal(I)}), c)]) \hookrightarrow \pi[i \rightarrow ((m[\texttt{I} \rightarrow n + 1], \texttt{nop}), c)]} \tag{11}$$

$$\frac{p' = \zeta(\texttt{P'})m \wedge p = \zeta(\texttt{P})m'}{\begin{array}{l}\pi[p \rightarrow ((m, \texttt{send(P',I)}), c)][p' \rightarrow ((m', \texttt{recv(P,I')}), c')] \hookrightarrow \\ \qquad \pi[p \rightarrow ((m, \texttt{nop}), c)][p' \rightarrow ((m'[\texttt{I'} \rightarrow m(\texttt{I})], \texttt{nop}), c')]\end{array}} \tag{12}$$

Fig. 11: `SPIRE(Stmt)` synchronized transition rules

The semantics of a whole parallel program `S` is defined as the set of memories m such that $\perp[0 \rightarrow ((\perp, \texttt{S}), \{\})] \hookrightarrow \pi[0 \rightarrow ((m, \texttt{nop}), c)]$, if `S` terminates.

5.3 Rewriting Rules

The SPIRE concepts not dealt with in the previous section are defined via their rewriting into the core language. This is the case for both the treatment of the `execution` attribute and the remaining coarse-grain synchronization constructs.

Execution. A parallel `sequence` of statements S_1 and S_2 is a pair of independent substatements executed simultaneously by spawned processes I_1 and I_2 respectively, i.e., is equivalent to:

```
barrier(spawn(I₁,S₁);spawn(I₂,S₂))
```

A parallel `forloop` (see Figure 3) with index `I`, lower expression `low`, upper expression `up`, step expression `step` and body `S` is equivalent to:

```
I=low;loop(I<=up,spawn(I,S);I=I+step)
```

A parallel `unstructured` is rewritten as follows. All control nodes present in the transitive closure of the successor relation are rewritten in the same manner. Each control node `C` is characterized by a statement `S`, predecessor list `ps` and successor list `ss`. For each edge (`c,C`), where `c` is a predecessor of `C` in `ps`, an event $I_{c,C}$ initialized at `newEvent(0)` is created, and similarly for `ss`. The whole unstructured construct is replaced by a sequential sequence of `spawn(I,`S_c`)`, one for each `C` of the transitive closure of the successor relation starting at the `entry` control node, where S_c is defined as follows:

```
barrier(spawn(1,wait(I_{ps[1],C}));...;spawn(m,wait(I_{ps[m],C})));
S;
signal(I_{C,ss[1]});...;signal(I_{C,ss[m']})
```

where `m` and `m'` are the length of the `ps` and `ss` lists; `L[I]` is the `I`-th element of `L`.

Synchronization. A statement `S` with synchronization `atomic(I)` is rewritten as:

```
wait(I);S;signal(I)
```

assuming that the assignment `I = newEvent(1)` is performed on the `event` identifier `I` at the very beginning of the whole program. A `wait` on an event variable sets it to zero if it is currently equal to one to prohibit other threads to enter the atomic section; the `signal` resets the event variable to one to permit further access.

A statement `S` with a blocking synchronization `single`, i.e., equal to `true`, is equivalent, when it occurs within an enclosing innermost parallel `forloop`, to:

```
barrier(wait(I_S);
        if(first_S,
           S; first_S = false,
           nop);
        signal(I_S))
```

where `first_S` is a boolean variable that ensures that only one process among those spawned by the parallel loop will execute `S`; access to this variable is protected by the event `I_S`. Both `first_S` and `I_S` are respectively initialized before loop entry to `true` and `newEvent(1)`. The conditional `if(E,S,S')` can easily be rewritten using the core `loop` construct. The same rewriting can be used when the `single` synchronization is equal to `false`, corresponding to a non-blocking synchronization construct, except that no `barrier` is needed.

6 Validation

Assessing the quality of a methodology that impacts the definition of a data structure as central for compilation frameworks as an intermediate representation is a difficult task. This section provides two possible ways to perform such an assessment on SPIRE: (1) we illustrate how it can be used on a different IR, namely LLVM, with minimal changes, thus providing support regarding the generality of our methodology, and (2) we provide information regarding its impact on run-time performance data for parallelization.

6.1 SPIRE Application to LLVM IR

This section shows how simple it is to upgrade the LLVM IR to a "parallel LLVM IR" via SPIRE transformation methodology.

LLVM [13] (Low-Level Virtual Machine) is an open source compilation framework that uses an intermediate representation in Static Single Assignment (SSA) [27] form. We chose the IR of LLVM to illustrate a second time our approach since LLVM has been widely used in both academia and industry; another interesting feature of LLVM IR, compared to PIPS's, is that it sports a graph approach, while PIPS is abstract syntax tree-based; each function is structured in LLVM as a control flow graph (CFG).

Figure 12 provides the definition of a significant subset of the sequential LLVM IR described in [13] (to keep notations simple in this paper, we use the same Newgen language to write this specification):

- a function is a list of basic blocks, which are portions of code with one entry and one exit points;
- a basic block has an entry label, a list of ϕ nodes and a list of instructions, and ends with a terminator instruction;
- ϕ nodes, which are the key elements of SSA, are used to merge the values coming from multiple basic blocks. A ϕ node is an assignment (represented here as a call expression) that takes as arguments an identifier and a list of pairs (value, label); it assigns to the identifier the value corresponding to the label of the block preceding the current one at run time;
- every basic block ends with a terminator which is a control flow-altering instruction that specifies which block to execute after termination of the current one.

```
function              = blocks:block*;
block                 = label:entity x phi_nodes:phi_node* x
                        instructions:instruction* x terminator;
phi_node              = call;
instruction           = call;
terminator            = conditional_branch + unconditional_branch + return;
conditional_branch    = value:entity x label_true:entity x label_false:entity;
unconditional_branch = label:entity;
return                = value:entity;
```

Fig. 12: Simplified Newgen definitions of the LLVM IR

Applying SPIRE to LLVM IR is, as illustrated above with PIPS, achieved in three steps, yielding the SPIREd parallel extension of the LLVM sequential IR provided in Figure 13:

- an `execution` attribute is added to `function` and `block`: a parallel basic block sees all its instructions launched in parallel (in a fork/join manner), while all the blocks of a parallel function are seen as parallel tasks to be executed concurrently;

– a `synchronization` attribute is added to `instruction`; therefore, an instruction can be annotated with `spawn`, `barrier`, `single` or `atomic` synchronization attributes. When one wants to deal with a sequence of instructions, this sequence is first outlined in a function, to be called instead; this new call instruction is then annotated by the proper synchronization attribute, such as `spawn`, if the sequence must be considered as an asynchronous task. A similar technique is used for the other synchronization constructs `barrier`, `single` and `atomic`;
– as LLVM provides a set of intrinsic functions [13], SPIRE functions `newEvent`, `signal` and `wait` for handling point-to-point synchronization, and `send` and `recv` for handling data distribution, are added to this set.

```
function′     = function x execution;
block′        = block x execution;
instruction′  = instruction x synchronization;
```

Fig. 13: SPIRE (LLVM IR)

Note that the use of SPIRE on the LLVM IR is not able to express parallel loops as easily as was the case on PIPS IR. Indeed, the notion of a loop does not always exist in the definition of intermediate representations based on control flow graphs, including LLVM; it is an attribute of some of its nodes, which has to be added later on by a loop-detection program analysis phase. Of course, such analysis could also be applied on the SPIRE-derived IR, and thus recover this information. Once one knows that a particular loop is parallel, this can be encoded within SPIRE using the same technique as presented in Section 5.3.

More generally, even though SPIRE uses traditional parallel paradigms for code generation purposes, SPIRE-derived IRs are able to deal with more specific parallel constructs such as DOACROSS or HELIX-like approaches. Basically, a compiler would parse a given sequential program into sequential IR elements. Optimization compilation phases specific to particular parallel code generation paradigms such as those above will translate, whenever possible (specific data and control-flow analyses will be needed here), these sequential IR constructs into parallel loops, with the corresponding synchronization primitives, as need be. Code generation will then recognize such IR patterns and generate specific parallel instructions such as DOACROSS.

6.2 Performance

We have used the SPIRE methodology to design a parallel IR for the PIPS infrastructure, and upgraded the PIPS implementation so that it can handle the resulting additional constructs, except for events, since our automatic parallelization algorithms do not require point-to-point synchronization for the moment. We have used this new version of PIPS for the implementation of a new task parallelization algorithm [28]. It automatically generates both OpenMP and MPI code from the same parallel IR.

We gathered performance measures related to SPIRE-based parallelization on four well-known C scientific applications, targeting both shared and distributed memory architectures: the image and signal processing applications Harris [29] and ABF [30], the SPEC2001 benchmark equake [31] and the NAS parallel benchmark IS [32]. Table 2 provides, for each application (Harris, ABF, equake and IS) and execution environment (OpenMP and MPI), the speedup obtained for runs of their SPIRE (PIPS)-coded parallel versions.

The execution times have been measured on two host Linux machines: (1) one with a 2-socket AMD quad-core Opteron with 8 cores and 16 GB of RAM, running at 2.4 GHz (when targeting OpenMP), and (2) another with 6 bicore processors Intel Xeon and 32 Gb of RAM per processor, running at 2.5 GHz (for MPI programs). The other details of this experimental protocol are provided in [28].

This set of experimental data suggests that the SPIRE methodology is able to provide parallel IRs that encode a significant amount of parallelism, and is thus well adapted to the design of parallel target formats for the efficient parallelization of scientific applications on both shared and distributed memory systems.

Application	Language	# of threads (OpenMP) or processes (MPI)	Speedup
Harris	OpenMP	3	2.21
	MPI	3	1.7
ABF	OpenMP	8	4.4
	MPI	6	2.8
equake	OpenMP	8	5.02
	MPI	6	3.05
IS	OpenMP	8	3.12
	MPI	6	2.86

Table 2. OpenMP/MPI vs. sequential speedup (Harris, ABF, equake and IS)

7 Conclusion

SPIRE is a new and general 3-step extension methodology for mapping any intermediate representation (IR) used in compilation platforms for representing sequential programming constructs to a parallel IR; one can leverage it for the source-to-source and high- to mid-level optimization of control-parallel languages and constructs.

The extension of an existing IR introduces (1) a parallel execution attribute for each group of statements, (2) a high-level synchronization attribute on each statement node and an API for low-level synchronization events and (3) two built-ins for implementing communications in message passing memory systems. The formal semantics of SPIRE transformational definitions is specified using a two-tiered approach: a small-step operational semantics for its base parallel concepts and a rewriting mechanism for high-level constructs.

The SPIRE methodology is presented via a use case, the intermediate representation of PIPS, a powerful source-to-source compilation infrastructure for Fortran and C. We illustrate the generality of our approach by showing how SPIRE can be used to represent the constructs of the current parallel languages Cilk, Chapel, X10, Habanero-Java, OpenMP, OpenCL and MPI. We provide experimental elements to validate SPIRE: (1) the application of SPIRE on another IR, namely the one of the widely-used LLVM compilation infrastructure, and (2) performance data, gathered using our implementation of SPIRE on PIPS IR, to illustrate the ability of the resulting parallel IR to efficiently express parallelism present in scientific applications.

Future work will address the representation via SPIRE of the PGAS memory model and of more programming features such as exceptions.

References

1. R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: An Efficient Multithreaded Runtime System," in *Journal of Parallel and Distributed Computing*, 1995, pp. 207–216.
2. B. Chamberlain, D. Callahan, and H. Zima, "Parallel Programmability and the Chapel Language," *Int. J. High Perform. Comput. Appl*, vol. 21, pp. 291–312, August 2007.
3. P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, "X10: An Object-Oriented Approach to Non-Uniform Cluster Computing," *SIGPLAN Not.*, vol. 40, pp. 519–538, October 2005.
4. V. Cavé, J. Zhao, and V. Sarkar, "Habanero-Java: the New Adventures of Old X10," in *9th International Conference on the Principles and Practice of Programming in Java (PPPJ)*, August 2011.
5. OpenMP, "Specifications," http://openmp.org/wp/openmp-specifications/.
6. OpenCL, "The Open Standard for Parallel Programming of Heterogeneous Systems," http://www.khronos.org/opencl.
7. MPI, "Message Passing Interface," http://www-unix.mcs.anl.gov/mpi.
8. J. Zhao and V. Sarkar, "Intermediate Language Extensions for Parallelism," in *Proceedings of the compilation of the co-located workshops on DSM'11, TMC'11, AGERE!'11, AOOPES'11, NEAT'11, &#38; VMIL'11*, ser. SPLASH '11 Workshops. New York, NY, USA: ACM, 2011, pp. 329–340. [Online]. Available: http://doi.acm.org/10.1145/2095050.2095103
9. S. Pai, R. Govindarajan, and M. J. Thazhuthaveetil, "PLASMA: Portable Programming for SIMD Heterogeneous Accelerators," in *Workshop on Language, Compiler, and Architecture Support for GPGPU, held in conjunction with HPCA/PPoPP 2010*, Bangalore, India, January 9, 2010.
10. Insieme, "Insieme - an Optimization System for OpenMP, MPI and OpenCL Programs," http://www.dps.uibk.ac.at/insieme/architecture.html.
11. F. Coelho, P. Jouvelot, C. Ancourt, and F. Irigoin, "Data and Process Abstraction in PIPS Internal Representation," in *Proceedings of the Workshop on Intermediate Representations*, F. Bouchez, S. Hack, and E. Visser, Eds., 2011, pp. 77–84.
12. F. Irigoin, P. Jouvelot, and R. Triolet, "Semantical Interprocedural Parallelization: An Overview of the PIPS Project," in *ICS*, 1991, pp. 244–251.
13. T. L. D. Team, *The LLVM Reference Manual (Version 2.6)*, http://llvm.org/docs/LangRef.html, 2010.
14. D. Novillo, "OpenMP and Automatic Parallelization in GCC," in *the Proceedings of the GCC Developers Summit*, June 2006.
15. J. Merrill, "GENERIC and GIMPLE: a New Tree Representation for Entire Functions," in *GCC developers summit 2003*, 2003, pp. 171–180.
16. M. Girkar and C. D. Polychronopoulos, "Automatic Extraction of Functional Parallelism from Ordinary Programs," *IEEE Trans. Parallel Distrib. Syst.*, vol. 3, pp. 166–178, March 1992.
17. Y. Choi, Y. Lin, N. Chong, S. Mahlke, and T. Mudge, "Stream Compilation for Real-Time Embedded Multicore Systems," in *Proceedings of the 7th annual IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO '09, Washington, DC, USA, 2009, pp. 210–220.
18. V. Sarkar and B. Simons, "Parallel Program Graphs and their Classification," in *LCPC*, ser. Lecture Notes in Computer Science, U. Banerjee, D. Gelernter, A. Nicolau, and D. A. Padua, Eds., vol. 768. Springer, 1993, pp. 633–655.
19. J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The Program Dependence Graph And Its Use In Optimization," *ACM Trans. Program. Lang. Syst.*, vol. 9, no. 3, pp. 319–349, Jul. 1987.
20. N. Benoit and S. Louise, "Kimble: a Hierarchical Intermediate Representation for Multi-Grain Parallelism," in *Proceedings of the Workshop on Intermediate Representations*, F. Bouchez, S. Hack, and E. Visser, Eds., 2011, pp. 21–28.
21. P. Jouvelot and R. Triolet, "Newgen: A Language Independent Program Generator," CRI/A-191, MINES ParisTech, Tech. Rep., July 1989.

22. E. W. Dijkstra, “Re: “Formal Derivation of Strongly Correct Parallel Programs” by Axel van Lamsweerde and M.Sintzoff,” 1977, circulated privately. [Online]. Available: http://www.cs.utexas.edu/users/EWD/ewd06xx/EWD631.PDF
23. D. Khaldi, P. Jouvelot, C. Ancourt, and F. Irigoin, “Task Parallelism and Data Distribution: An Overview of Explicit Parallel Programming Languages,” in *Proceedings of the 2012 International Workshop on Languages and Compilers for Parallel Computing*, ser. LCPC’12, 2012.
24. E. W. Dijkstra, “Cooperating Sequential Processes,” 1968, published as [33]. [Online]. Available: http://www.cs.utexas.edu/users/EWD/ewd01xx/EWD123.PDF
25. K. Yelick, D. Bonachea, W.-Y. Chen, P. Colella, K. Datta, J. Duell, S. L. Graham, P. Hargrove, P. Hilfinger, P. Husbands, C. Iancu, A. Kamil, R. Nishtala, J. Su, W. Michael, and T. Wen, “Productivity and Performance Using Partitioned Global Address Space Languages,” in *Proceedings of the 2007 International Workshop on Parallel Symbolic Computation*, ser. PASCO ’07. New York, NY, USA: ACM, 2007, pp. 24–32.
26. T. U. Consortium, “UPC Specification,” http://upc.gwu.edu/documentation.html, June 2005.
27. R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, “Efficiently Computing Static Single Assignment Form and the Control Dependence Graph,” *ACM Trans. Program. Lang. Syst.*, vol. 13, no. 4, pp. 451–490, Oct. 1991. [Online]. Available: http://doi.acm.org/10.1145/115372.115320
28. D. Khaldi, P. Jouvelot, and C. Ancourt, “Parallelizing with BDSC, a Resource-Constrained Scheduling Algorithm for Shared and Distributed Memory Systems,” MINES ParisTech, Tech. Rep. CRI/A-499 (Submitted to *Parallel Computing*), 2012.
29. C. Harris and M. Stephens, “A Combined Corner and Edge Detector,” in *Proceedings of the 4th Alvey Vision Conference*, 1988, pp. 147–151.
30. L. Griffiths, “A Simple Adaptive Algorithm for Real-Time Processing in Antenna Arrays,” *Proceedings of the IEEE*, vol. 57, pp. 1696 – 1704, 1969.
31. H. Bao, J. Bielak, O. Ghattas, L. F. Kallivokas, D. R. O’Hallaron, J. R. Shewchuk, and J. Xu, “Large-scale Simulation of Elastic Wave Propagation in Heterogeneous Media on Parallel Computers,” *Computer Methods in Applied Mechanics and Engineering*, vol. 152, no. 1–2, pp. 85–102, 1998.
32. NPB, “NAS Parallel Benchmarks,” http://www.nas.nasa.gov/publications/npb.html.
33. E. W. Dijkstra, “Cooperating Sequential Processes,” in *Programming Languages: NATO Advanced Study Institute*, F. Genuys, Ed. Academic Press, 1968, pp. 43–112.