SPIRE: A Methodology for Sequential to Parallel Intermediate Representation Extension

Dounia Khaldi, Pierre Jouvelot, François Irigoin, Corinne Ancourt

SPIRE: A 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 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 finally assess the generality of our proposal by showing how key parallel features of these current parallel languages can be dealt with using SPIRE.

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;**

I. Introduction

The growing importance of parallel computers and the search for an efficient programming model led, and is still leading, to the 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 II) 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 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 1 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 internal representation (IR) [11] of PIPS [12], a comprehensive source-to-source compilation and optimization platform.

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. To assess the validity of the generic parallel IR we designed and present here, we benchmarked it against key parallel languages and showed 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 easily leverages their existing infrastructure for sequential languages to address both control and data parallelism and data distribution;
- a parallel IR, to be used for both automatic task-level parallelization and the optimization of explicitly parallel programs, for the PIPS compilation framework;
- an evaluation of the generality of SPIRE, by mapping key parallel programming languages, i.e., Cilk, Chapel, X10, Habanero-Java, OpenMP, OpenCL and MPI, to it;
- the small-step, operational semantics of SPIRE, to formally defines its key parallel concepts.

After this introduction, we survey existing parallel IRs in Section II. We describe our use-case sequential IR, used by the PIPS compilation framework, in Section III. Our parallel extension proposal, SPIRE, is introduced in Section IV, 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 V. Section VI shows the generality of SPIRE by showing how different parallel language constructs can be mapped to it, and discusses implementation issues. We discuss future work and conclude in Section VII.

II. 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 what we believe are the key existing IRs addressing issues similar to our paper's. Bird-view comparisons with SPIRE are also given here, although a more detailed and admittedly useful analysis would require more space than permitted by the paper format.

A standard approach to parallelism expression is to use built-in functions. For instance, the intermediate representation of the implementation of OpenMP in GCC (GOMP) [13] extends its three-address representation, GIMPLE. The OpenMP parallel directives are replaced by specific nodes and built-ins, such as the `__sync_fetch_and_add` built-in function for an atomic memory access addition. SPIRE uses some of these ideas, but frames them in more structured settings while trying to be more language-neutral. Applying the SPIRE approach to GCC would have provided a minimal set of extensions that could have also be used for other parallel languages such as Cilk that relies on GCC as a back end; we illustrate our approach in the paper via PIPS in lieu of GCC.

Sarkar and Zhao [8] introduce the high-level parallel intermediate representation HPIR that is decomposed into a RST (region syntax tree), a region control-flow graph (RCFG) and a region dictionary (RD). Each parallel program construct is annotated accordingly: `AsyncRegionEntry` and `AsyncRegionExit` delimit a task, while the instructions `FinishRegionEntry` and `FinishRegionExit` can be used in parallel sections. The same comments as those mentioned for GCC apply here too.

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-independant and also covers control parallelism.

InsPIRe is the parallel intermediate representation at the core of the source-to-source Insieme compiler [10] for C++ parallel programs. SPIRE intends to also cover source-to-source optimization, but not for a single language only.

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 [14] 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 in a hierarchical fashion, thus enabling recursively-defined optimization techniques to be applied easily.

A stream graph [15] is a dataflow representation introduced specifically for streaming languages. Nodes represent data reorganization 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.

The parallel program graph [16] extends the program dependance graph [17], a directed graph 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. SPIRE adopts a similar extension approach to an existing sequential intermediate representation, but extends it to both structured and unstructured constructs in a hierarchical manner.

III. 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. PIPS is a powerful source-to-source compilation and optimization platform; its internal 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 internal representation of PIPS; it is specified using Newgen [18], 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 internal 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 "$\star$" 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 internal 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*;
```

Figure 1: Simplified Newgen definitions of the PIPS IR

IV. SPIRE, A SEQUENTIAL TO PARALLEL IR EXTENSION

We describe in this section how parallel concepts can be readily introduced into a sequential IR using our SPIRE approach. The core idea is that, to be able to deal with parallel programming, one 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 single statements and (3) the layout of data. The design of SPIRE does not intend to be minimalist but, using as input an extensive survey of existing parallel language constructs [19], to provide a trade-off between expressibility and conciseness of representation. In our PIPS IR case, SPIRE amounts to adding three new concepts: the `execution` set, the `synchronization` set and event API for synchronization purposes and the ability to handle data distribution through different memory models. We illustrate the application of SPIRE on the PIPS IR below.

A. 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 this IR, one simply needs to add an `execution` attribute 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, one only needs to add an execution field 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 to for all loop iterations, sequence statements and control nodes of unstructured instructions to be run concurrently.

For instance, a parallel execution construct can be used to represent the OpenCL `clEnqueueNDRangeKernel` function which implements data parallelism on GPUs (see Figure 2); here the `kernel` is executed in a parallel loop, each task receiving the proper index value as an argument.

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

Figure 2: OpenCL example illustrating a parallel loop

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)
```

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

B. 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;
```

where 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);
```

Figure 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 SPIRE statement accessing Array `x` can be translated in Cilk (via `Cilk_lock` and `Cilk_unlock`) and OpenMP (`atomic`) is provided in Figure 5.

C. Event API

In parallel code, one usually distinguishes between two types of synchronization: (1) coarse grain (collective) syn-

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

```
#pragma omp atomic
  x[index[i]] += f(i);
```

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

chronization between threads using barriers, which are handled by SPIRE using the `synchronization` patterns above, and (2) fine grain (point-to-point) synchronization between participating threads. Handling point-to-point synchronization using decorations on abstract syntax trees 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. The programming interface for events is defined by the following atomic 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[1] of `e`;
- `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 tasks is completed and `wait`ed by the `force` method.

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

Figure 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`. The `next` statement is equivalent to:

```
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)))
```

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

D. 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 new 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, 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`.

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

[1]The `void` return type will be replaced by `int` in practice, to enable the handling of error values.

```
MPI_Init(int argc, char *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();
```

Figure 8: MPI example illustrating a communication

An other interesting memory model for parallel programming has been introduced somewhat recently: the Partitioned Global Address Space [20]. The uses of the PGAS memory model in languages such as 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.

V. 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, 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: (1) we introduce 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.

A. Sequential Core Language

Illustrating the transformation induced by SPIRE requires the definition of a sequential IR basis. We use here, as sequential core language, `Stmt`, which includes the essentials from the PIPS sequential IR provided in Section III, where the information present in the `instruction` and `statement` sets are merged. The syntax of `Stmt` is given[2] as Line (*) in Figure 9, where we assume that the sets `Ide` of identifiers `I` and `Exp` of expressions `E` are given.

[2]For lack of space, we leave out the simple, but convoluted, management of sequential unstructured instructions.

```
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')
```

Figure 9: `Stmt` and `SPIRE(Stmt)` syntaxes

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

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 $n \in N$ 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'`. 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.

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

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

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

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

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

Figure 10: `Stmt` sequential transition rules

B. 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

[3]A PIPS `forloop` can be rewritten using a `loop` statement.

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 V-C). 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 $e \in \mathit{Event} = N$, 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 A sequential language uses a unique memory. 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 t and t' with $\pi(t) = ((m, \texttt{S}), c)$ and $\pi(t') = ((m', \texttt{S'}), c')$, one has[4] $m = m'$. No such constraint is needed for the message passing model. As mentioned above, PGAS is left for future work, where some sort of constraints based on the semantic definitions of 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.

Rule 6 bridges the sequential and the SPIRE-extended parallel semantics; note that the interleaving between parallel processes in `SPIRE(Stmt)` is a consequence of the non-deterministic choice of the value of i within $dom(\pi)$ when selecting the transition to perform and of the number of steps executed by the sequential semantics. In Rule 7, `spawn` adds a new process n that executes `S` to the state; 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'`.

C. 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 $\texttt{S}_1$ and $\texttt{S}_2$ is a pair of independent substatements executed simultaneously by spawned processes $\texttt{I}_1$ and $\texttt{I}_2$ respectively, i.e., is equivalent to:

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

A parallel `forloop`, an example of which appears in 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 $\texttt{I}_{\texttt{c,C}}$ initialized at `newEvent(0)` is created, and similarly for `ss`. The whole unstructured construct is replaced by a sequential sequence of `spawn(I,`$\texttt{S}_c$`)`, one for each `C` of the transitive closure of the successor relation starting at the `entry` control node, where $\texttt{S}_c$ is defined as follows:

[4]The issue of private variables in threads would have to be introduced in full-fledged languages.

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

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

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

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

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

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

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

Figure 11: `SPIRE(Stmt)` synchronized transition rules

```
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)` rewrites 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 enclosed 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.

VI. Validation

This section provides information on the practical use and benefits of SPIRE: (1) we illustrate how high-level parallel constructs used in the current parallel programming languages addressed in this paper can be translated using SPIRE concepts and (2) address implementation issues.

A. Mapping SPIRE to Parallel Programming Languages

Table I, extended from [19], summarizes the main characteristics of the parallel languages of interest in this paper: 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. We extend this table to include a line for MPI and another one which corresponds to the approach we suggest to use to map these concepts to our parallel intermediate representation

SPIRE, introducing only ten key notions, collected in three groups (execution, synchronization and data distribution): `sequential`, `parallel`, `spawn`, `barrier`, `atomic`, `single`, `signal`, `wait`, `send` and `recv`. We sketch below how the mapping of parallel languages to SPIRE can be implemented, in practice.

Task creation (e.g., `clEnqueueTask` in OpenCL, see Figure 4) defines a task code fragment to be executed in parallel with the outside task that creates it; the corresponding processes or threads join, later on, using task join synchronization primitives such as `finish` in X10 (see Figure 7). These operations can be mapped in SPIRE using the synchronization attribute `spawn` for the task creation statement while selecting the `barrier` synchronization for the outside task.

An other form of synchronization, finer than task join, is point-to-point synchronization, where the affected tasks use event variables for such a purpose (see phasers in Habanero-Java Figure 7); SPIRE uses the two built-ins `signal` and `wait` to translate these events.

An atomic section implements mutual exclusion (e.g., `Cilk_lock` and `Cilk_unlock` in Cilk and `atomic` in OpenMP, see Figure 5) is mapped in SPIRE using the synchronization attribute `atomic`.

Data parallelism (e.g., `clEnqueueNDRangeKernel` in OpenCL, see Figure 2), where the same operation is applied repeatedly to different items, is represented in SPIRE using the execution attribute `parallel` to a `forloop` statement.

In addition to the previous control constructs, the table specifies each language data-distribution memory model. SPIRE includes two models: shared (the default model) and message passing. In the shared model, private variables are represented as local variables, initialized via copy operations. For message passing (e.g., `send` and `recv` in MPI, see Figure 8), SPIRE offers the two built-ins necessary for explicit communication: `send` and `recv`. For the time being, we consider PGAS code as shared, a PGAS implementation in SPIRE being left as future work.

B. Implementation

We have implemented the SPIRE-derived parallel IR presented above[5] in the PIPS middle-end, and used it for the implementation of a new BDSC-based task parallelization algorithm [21]. We generate both OpenMP and MPI code from the same parallel IR. This first implementation suggests that our main goal with the design of SPIRE, namely the reuse of existing software developments, is reachable; indeed, we were able to easily leverage some of the sequential optimizations present in the original PIPS platform, a key economic advantage.

Even though we used traditional parallel paradigms for code generation purposes, we believe that 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.

[5]Events have been omitted since our automatic parallelization algorithms do not address point-to-point synchronization issues.

VII. Conclusion

SPIRE is a new and general 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. As a use case for the introduction of SPIRE, we describe the extension of 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.

Future work will address the representation of the PGAS memory model in SPIRE and the implementation of transformations for parallel languages encoded in SPIRE.

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.

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

Table I
MAPPING OF SPIRE TO PARALLEL LANGUAGES CONSTRUCTS (TERMS IN PARENTHESES ARE NOT CURRENTLY HANDLED BY SPIRE)

[5] OpenMP, http://openmp.org/wp/openmp-specifications/.

[6] OpenCL, "The Open Standard for Parallel Programming of Heterogeneous Systems," http://www.khronos.org/opencl.

[7] MPI, http://www-unix.mcs.anl.gov/mpi.

[8] J. Zhao and V. Sarkar, "Intermediate Language Extensions for Parallelism," in *The 5th Workshop on Virtual Machines and Intermediate Languages*. New York, NY, USA: ACM, 2011.

[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/mission.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] D. Novillo, "OpenMP and Automatic parallelization in GCC," in *the Proceedings of the GCC Developers Summit*, June 2006.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] P. Jouvelot and R. Triolet, "Newgen: A Language Independent Program Generator," CRI/A-191, MINES ParisTech, Tech. Rep., July 1989.

[19] 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.

[20] 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.

[21] 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.