A Parallel Intermediate Representation
— Draft —

Johannes Doerfert Simon Moll Kevin Streit
Sebastian Hack

September 29, 2016

Contents

1 Introduction

Extending an existing compiler intermediate language with parallel constructs is a challenging task. While maintainability dictates a minimal extension that will not disturb to many of the existing analyses and transformations it is also important to make the parallel constructs powerful enough to express different, orthogonal execution scenarios. For C/C++ *OpenMP* is one of the most prominent parallelization frameworks, however even OpenMP allows for multiple parallelization schemes (ref. Section 4.1). Additionally, other language extensions such as *OpenCL* might profit from the translation to lower level parallel constructs. Finally, automatic parallelizers and new (partially) parallel languages can be utilized best with general parallel constructs that allow to express parallel (or better concurrent) execution in a independent and intuitive way.

1.1 Goals

Minimal
: The additions to the IR and especially to the analysis and transformation passes present (and to be developed) should be minimal.

Robust
: The parallel semantics should be robust under current/common and future optimizations.

Intuitive
: The Parallel-IR should be as intuitive as sequential IR.

Expressive
: Existing parallel languages or language extensions as well as current and future parallel paradigms should be expressible.

Independent
: The Parallel-IR should not be coupled to a fixed set of parallel languages, language extensions or paradigms.

1.2 Common Problems

Integration
: Overloading of the sequential IR (e.g., instructions become context sensitive) or the CFG (e.g., implicit representation of multiple, concurrent execution traces).

Specialization
: The parallel constructions are suited for one language (e.g., OpenMP) or parallel paradigm (loop parallelism) but nothing else.

Fragility
: Optimizations have to be turned off completely or thought to handle the parallel parts differently (or not at all).

Plain Hacks
: Memory operations are marked **`volatile`** to prevent optimizations that *currently* bail on such instructions.

2 IR Extensions

To amend the existing LLVM-IR with parallel semantics we introduce two new terminator instructions, `fork` and `halt`, as well as a landingpad like `join` instruction. The `halt` terminator will stop the execution of the reaching thread, thus it has no successor in the Parallel-CFG. The `fork` terminator has successor blocks but it is different to classical terminators like **`br`** (branch) and **`switch`**. While classically control is transfered to exactly one successor, `fork` will transfer control to *all* successors. Consequently, Parallel-CFG semantics are defined with regards to DAG shaped execution traces that can be linearized (e.g., by the hardware) in various ways. This is a major difference to the CFG semantics used nowadays and needs to be communicated as such.

2.1 The **`fork`** Terminator

```
fork [force] [width <%w>] [lockstep] [label <master>]
     \[label <d1>, ... \]
fork interior \[<type> <%v1>, ... \] [label <master>]
     \[label <d1>, ... \]
```

Figure 1. `fork` syntax

The `fork` instruction is only valid if it has at least one successor block. As indicated in the syntax, one can specify a special successor block that is guaranteed to be executed by the "*master*" thread, hence the thread that reached the `fork` instruction. All other successors can be executed by any thread. One extreme would be to sequentialize the parallel region completely, hence to execute all successors with only one thread in a statically fixed order. However, this is only valid if the `force` attribute is not set. If it is, optimizations are not allowed to limit the possible execution traces statically, e.g., through sequentialization.

The value following the `width` attribute defines an upper bound on the number of threads, or more generally, concurrently executed parts in the parallel region. It allows to express dependence distances for loops (e.g., for the SIMD case) but can also restrict the amount of parallelism the compiler is supposed to exploit.

The `lockstep` attribute indicates that the successors of `fork` instructions in this parallel region have to be executed in lockstep fashion. This attribute is solely used to represent exact SIMD semantics.

An attribute that is present at at least one `fork` instruction in a parallel region causes the same semantics as if it was present at all `fork` instructions in the parallel region.

Interior `fork` instructions have the same semantics as regular `fork` instructions, except that they do not start a new parallel region but are part of the enclosing one. The program is not valid if no parallel region surrounds the `fork interior` instructions. The main (but not onliest) purpose of the

`fork interi`**`or`** instruction is to allow parallel loop constructs. As each parallel region has to contain at least one regular `fork` instruction it suffices to allow attributes only for them. However, `fork interi`**`or`** instructions can alternatively reference values to make them users of these values. This is a "hack' to prevent hoisting of e.g., private memory allocated in a parallel loop (see Figure 6 or Figure 8) and does not bear any other semantics.

Lastly, metadata attached to the `fork` instructions indicates the parallel backends that can be used to implement the `fork`. In contrast to most metadata this bears semantics. An example annotation could look like

```
%0 = !{ !"sequential", !"openmp"}
```

and it would force the backend to either sequentialize the parallel region or use OpenMP runtime calls to implement it. Similar to code backends, the compiler is informed at its own compile time of the parallel backends supported on the target machine. If no annotations are given for a parallel regions it is free to choose from this set. As a fall-back a built-in parallel runtime should be provided.

2.2 The `join` Instruction

```
join
```

Figure 2. `join` syntax

The `join` instruction is a landingpad style instruction, thus has to be placed at the beginning of a basic block. Furthermore, such basic blocks should not contain any **`phi`** nodes. The semantics of a `join` is twofold. First, it defines the end of a parallel region and second, the `join` instructions of a region form an implicit barrier for all concurrently running threads spanned in this region.

2.3 The `halt` Terminator

```
halt
```

Figure 3. `halt` syntax

The `halt` instruction will terminate the executing thread gracefully, thus it does not have any successor blocks in the Parallel-CFG. In contrast to the `join` instruction it does not perform any syncronization. It is also not possible to communicate scalars out of a thread that executed a `halt` instruction. Nevertheless, memory allows for communication and syncronization.

2.4 Common intrinsics

While most parallel languages and frameworks are defined from scratch and with different concepts in mind, most share a common core of built-in or library functions. While we do not try to replace every last library call present in language extensions like OpenMP or OpenCL in the IR, we represent the common core shared between all of them. To this end, the front-end lowers such calls to `llvm.parallel.XXX` intrinsic calls that are understood by the analysis and optimization passes. Other built-in function or library calls will be translated to regular calls with unknown side-effects, thus implicitly block optimizations.

Candidates for `llvm.parallel` intrinsics are shown in Figure 4. The first intrinsic returns the number of threads executing concurrently while the second returns a unique id for the calling thread.

```
declare i32 @llvm.parallel.num_threads()
declare i32 @llvm.parallel.thread_id()
```

Figure 4. `llvm.parallel` instrinsics

3 Parallel Regions

Parallel Regions are the abstract construct that specify both, validity and semantic of a Parallel-CFG. A parallel region θ is comprised of entry `fork` instructions $\mathbb{F}_{\mathbb{E}}$, `fork interi`**`or`** instructions $\mathbb{F}_{\mathbb{I}}$, `join` instructions $\mathbb{J}$ and `halt` instructions $\mathbb{H}$. The well-formedness criterion for Parallel-CFGs is given in Definition 3 and parallel region are formally introduced in Definition 4. As a consequence we can show the *Perfect-Nesting* Theorem 3.

Definition 1 (Parallel Nesting Depth)**.**
The parallel nesting depth τ is an inductive predicate over all finite trace prefixes. It defines how many parallel regions surround a basic block.

$$\tau(p_0) = 0$$
$$\tau(p_0, \ldots, p_n) = \tau(p_0, \ldots, p_{n-1}) + \#(term(p_n))$$

Where the term-function yields the terminator of a basic block. The #-function is used to increase the nesting depth for entry `fork` instructions and to decrease it for `join` and `halt` instructions.

$$\#(t) = \begin{cases} 1 & \text{if } t \in \mathbb{F}_{\mathbb{E}} \\ -1 & \text{if } t \in \mathbb{J} \cup \mathbb{H} \\ 0 & \text{otherwise} \end{cases}$$

Definition 2 (Parallel-CFG Consistency)**.**
A Parallel-CFG is consistent, if the parallel nesting depth τ of each block is the same for all paths reaching the block.

Definition 3 (Parallel-CFG Well-formedness)**.**
*A Parallel-CFG is well-formed, if (1) it is consistent, (2) the parallel nesting depth τ is always non-negative, and (3) τ is positive for each block terminated by an `fork interi`**`or`** instruction.*

Definition 4 (Parallel Region)**.**
Each maximal, weakly connected subgraph of a Parallel-CFG is a parallel region *θ_l if the parallel nesting depth τ of each block is at least $l \in \mathbb{N}^+$.*

Theorem 1 (Perfect Depth Nesting)**.**
For each parallel region θ_l with minimal parallel nesting depth $l > 1$ there exists a parallel region θ_{l-1} with minimal parallel nesting depth $l - 1$ that is a strict superset of θ_l.

Theorem 2 (Perfect Global Nesting)**.**
The set of blocks in parallel regions with minimal parallel nesting depth l is a strict subset of the set of the set of blocks in parallel regions with minimal parallel nesting depth l' for $l > l' > 0$.

Theorem 3 (Perfect Region Nesting)**.**
Parallel regions are perfectly nested, hence for every two parallel regions θ^1 *and* θ^2 *either they are independent or one is a subregion of the other:*

$$\theta^1 \cap \theta^2 \in \{\theta^1, \theta^2, \emptyset\}$$

Theorem 4 (Parallel Region Forest)**.**
The parallel regions of a function form a forest. If an auxiliary root node is introduced that spans the whole function, the parallel regions and this auxiliary root form a tree.

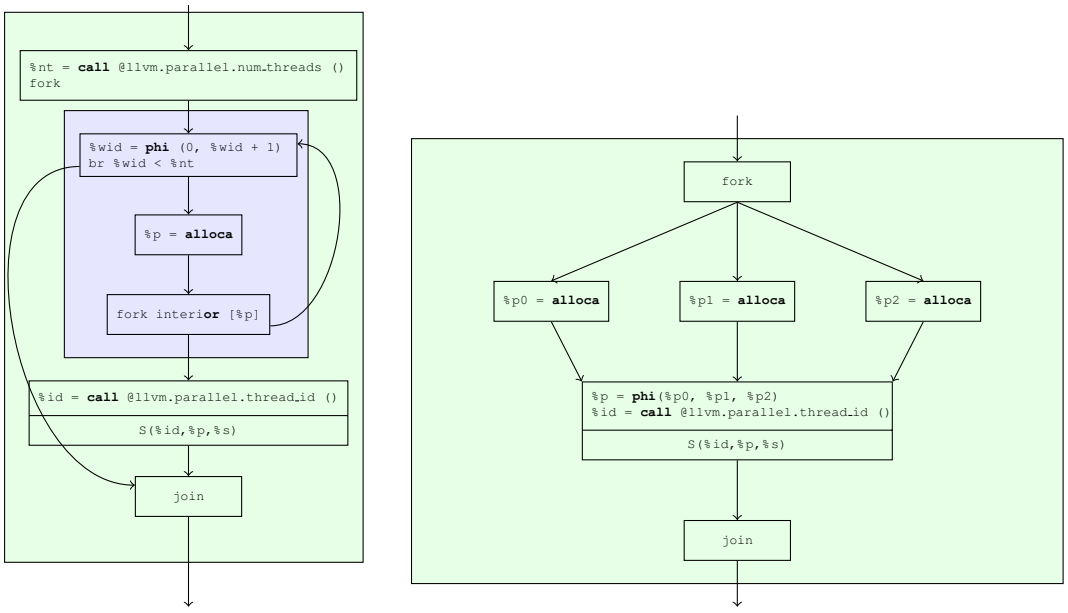
4 High-Level-Lowering

4.1 OpenMP

4.1.1 #pragma omp parallel

```
#pragma omp parallel private(p) shared(s)
{
  int id = omp_get_thread_num();
  S(id, p, s)
}
```

Figure 5. Generic OpenMP parallel clause



(a) Parallel-CFG for Figure 5

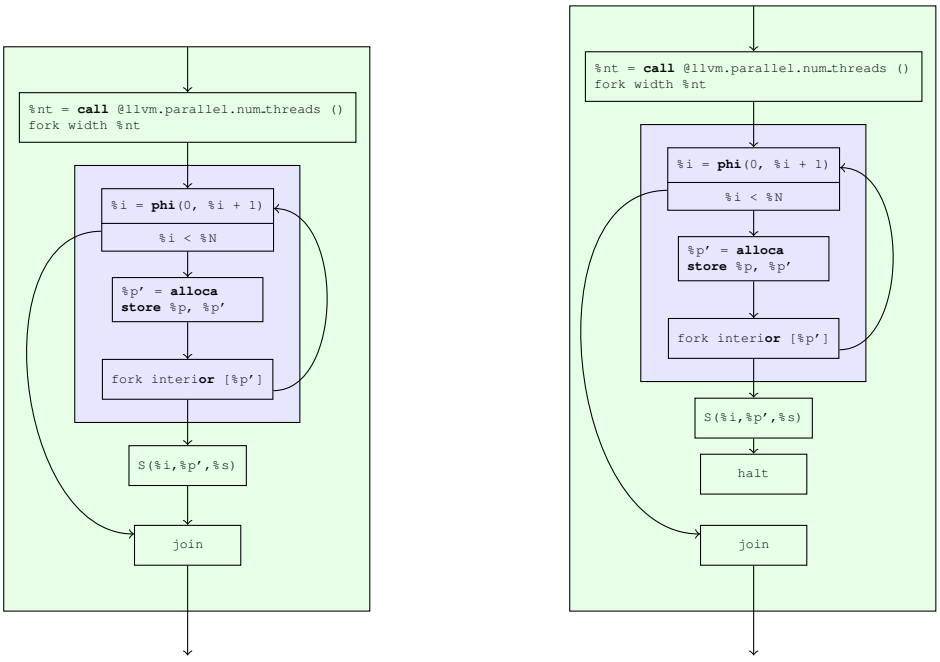
(b) Parallel-CFG for Figure 5 assuming an additional `num_threads(3)` attribute

Figure 6. Parallel-CFG for OpenMP parallel clause

4.1.2 #pragma omp parallel for

```
#pragma omp parallel for firstprivate(p) shared(s)
for (int i = 0; i < N; i++)
  S(i, p, s)
```

Figure 7. Simple OpenMP parallel for loop



(a) Parallel-CFG for Figure 7

(b) Parallel-CFG for Figure 7 assuming an additional `nowait` attribute

Figure 8. OpenMP Parallel For Loop

4.1.3 #pragma omp simd

```
#pragma omp simd private(p), safelen(n)
for (int i = 0; i < N; i++)
  S(i, p)
```

Figure 9. Simple OpenMP SIMD loop

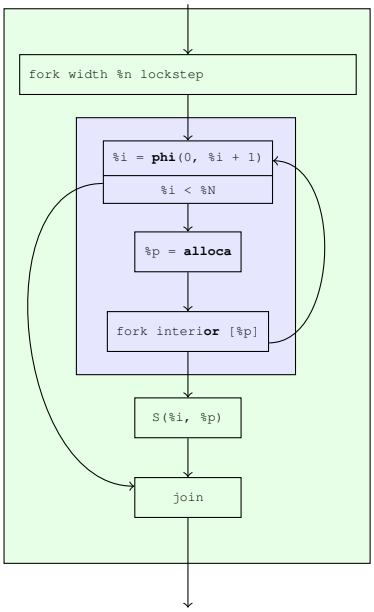


Figure 10. OpenMP SIMD Loop

4.2 Cilk+

4.2.1 `cilk_for` loops

These can be translated similar to parallel OpenMP loops (ref. Section 4.1.2) without other annotations.

4.2.2 `cilk_spawn` & `cilk_sync`

Translate (almost) directly into plain `fork` and `join` instructions with additional `join` insturctions prior to return statements.

```
__global__ void  ChildKernel(void* data) {
  // Operate on data
}
__global__ void ParentKernel(void *data) {
  if (...)
    ChildKernel<<<16, 1>>>(data);
  // Operate on data
}

ParentKernel<<<256, 64>>>(data);
```

Figure 11. CUDA snippet with dynamic parallelism

4.3 CUDA

The translation of CUDA to Parallel-IR is very similar to the translation of OpenCL to Parallel-IR. We therefore refer the reader to Section 4.4 for more examples. However, in contrast to OpenCL, CUDA allows for dynamic parallelism. That means a kernel can dynamically spawn another kernel or itself recursively. A simple example for dynamic parallelism is illustrated in Figure 11 and the corresponding Parallel-CFG is shown in Figure 12. It is important to note the similarities between the Parallel-IR representation of these two 1D CUDA kernels and the the 2D OpenCL example shown in Figure 17 and 18. The main difference is the arbitrary code and control in the outer parallel region that may or may not lead to the execution of the inner parallel region.

4.3.1 CUDA Thrust

CUDA Thrust is a library for transforming data structures in parallel on GPUs. In the code snippet Figure 13, the element-wise negation of one device array is stored in another one. This transformation can be lowered to Parallel-IR as in Figure 14. Note how Parallel-IR completely captures the parallel execution and data transformation on the arrays. Except for memory allocations, no calls to the CUDA library are necessary.

4.3.2 Fusing CUDA Thrust transformations

Parallel-IR completely captures the semantics of Thrust transformations in parallel loops. This enables the compiler to fuse sequences of Thrust transformations without special knowledge about Thrust but only arguing about (parallel) loops. The simplest example with a sequence of two parallel transformations (`sequence` and `transform`) is shown in Figure 15. In Figure 16, the Parallel-CFG for this code snippet is shown together with the optimized, fused version.

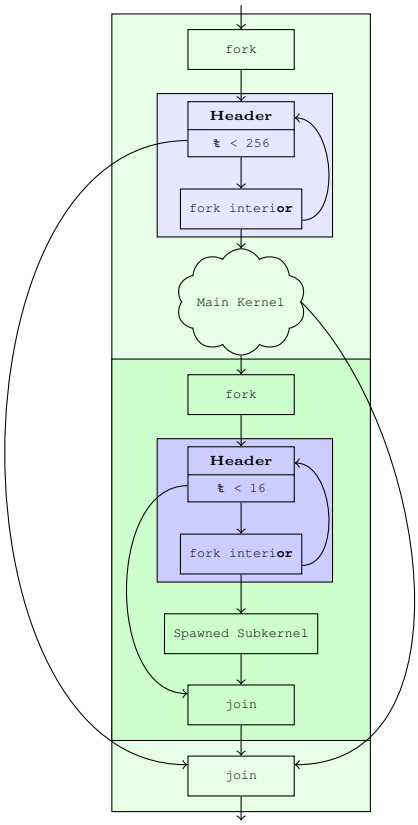


Figure 12. Parallel-CFG for the CUDA kernel with dynamic parallelism shown in Figure 11.

```
thrust::device_vector<int> X(10);
thrust::device_vector<int> Y(10);
thrust::transform(Y.begin(), Y.end(), X.begin(),
                  thrust::negate<int>());
```

Figure 13. CUDA Thrust library calls

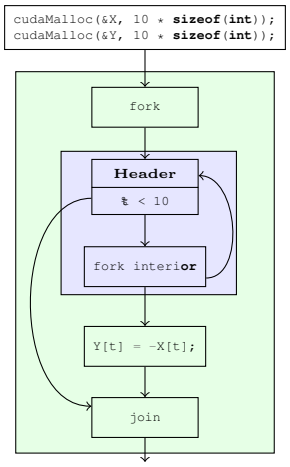
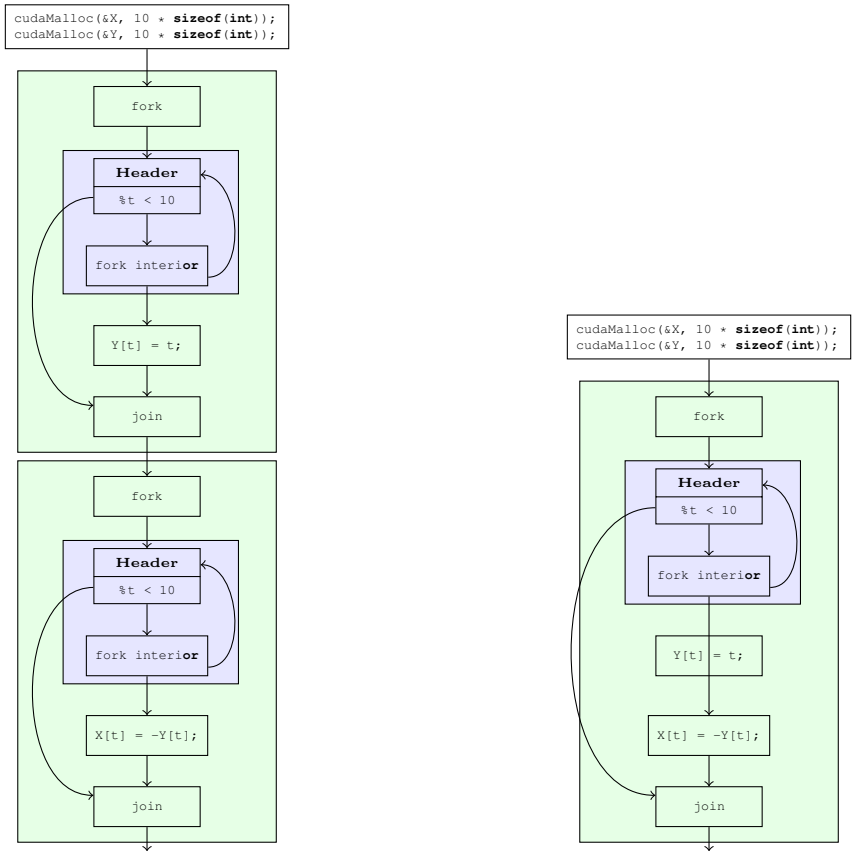


Figure 14. Parallel-CFG for the CUDA thrust snippet in Figure 13

```
thrust::device_vector<int> X(10);
thrust::device_vector<int> Y(10);
thrust::sequence(Y.begin(), Y.end(), 1); // {1, 2, ..}
thrust::transform(Y.begin(), Y.end(), X.begin(),
                  thrust::negate<int>()); // {-1, -2,  ..}
```

Figure 15. A sequence of Thrust library calls



(a) Parallel-CFG for Figure 15.

(b) Parallel-CFG after parallel loop fusion was applied to Figure 16a.

Figure 16. Parallel loop fusion applied to two CUDA Thrust library calls that were lowered to Parallel-IR.

4.4 OpenCL

OpenCL kernels can be represented with part of the driver code in the Parallel-IR to allow various loop transformations. One possible lowering of the OpenCL kernel shown in Figure 17 is illustrated in Figure 18. Except the actual kernel it also contains a parallel loop over the work groups in dimension 0 and another parallel loop over the elements of each work group. This way local and global memory can be represented in the IR too.

```
__kernel void simpleKernel(__global float * A) {
  int g = get_group_id(0);
  int t = get_local_id(0);
  __local int p[128];
  /* kernel */ S(A, g, t, p);
}
```

Figure 17. Generic OpenCL kernel

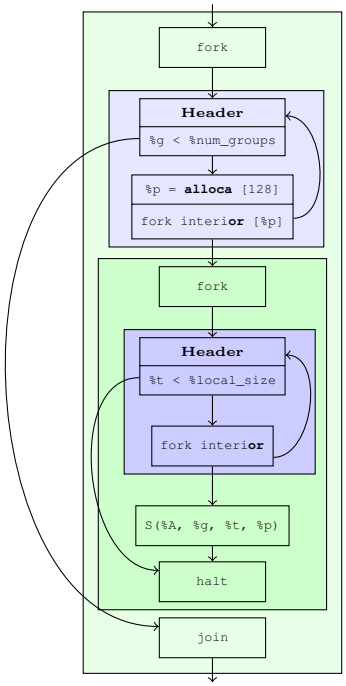


Figure 18. Parallel-CFG for OpenCL kernel in Figure 17

4.5 General Thread Level Parallelism

While general thread-level parallelism (`std::thread` or pthreads) is not in the scope of this parallel IR, some common patterns can be represented and optimized. However, if the creation or termination of threads is guarded by different control conditions, distributed over multiple functions or not perfectly nested, we avoid low level IR representation and stick with library calls.

The example in Figure 19 can be translated into the Parallel-CFG shown in Figure 20 while the example in Figure 21 is not ammendable to our representation.

```
pthread_create(&tid, nullptr, f, (void *)&v);
S(); /* Code without return or exceptions */
pthread_join(&tid);
```

Figure 19. Representable pthreads calls

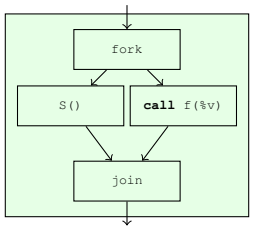


Figure 20. Parallel-CFG for pthreads calls in Figure 19

```
for (i = 0; i < N; i++)
  pthread_create(&tids[i], nullptr, f, (void *)&v[i]);

/* some code */

for (i = 0; i < N; i++)
  pthread_join(&tids[i]);
```

Figure 21. Non-representable pthreads calls

5 Transformations (notes only)

5.1 Parallel Region

Theorem 5. *Empty Parallel Regions An empty parallel region contains only* `fork` *and* `join` *instructions. Empty parallel regions can be removed.*

5.2 Subregion Merging

If all entry `fork`s of a parallel region θ are in starting blocks of the parent parallel region θ' and all `join`s of θ are in the exit blocks of θ', θ can be merged into θ'.

5.3 Disjunct Region Merging

5.4 Sequentializing Parallel Regions

Theorem 6 (Sequential Parallel Region)**.**
A parallel region is actually sequential if there is no `fork` *that has more than one successor.*

5.5 `fork` Transformations

Theorem 7 (Trivial Interior `fork`s)**.**
An interior `fork` f_i *with exactly one predecessor can be replaced by a unconditional branch to that predecessor. The other direction is also valid, however the parallel region that is associated with the new interior fork needs to contain the original unconditional branch.*

5.6 `join` Transformations

6 Analyses And Optimizations

6.1 Dominance

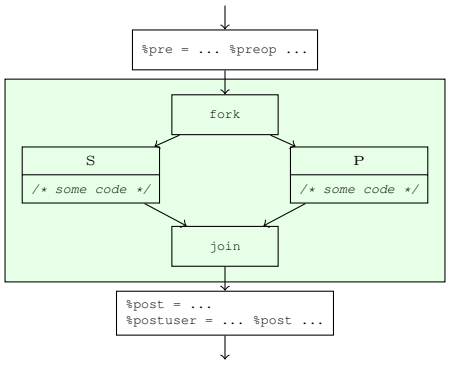


Figure 22. Sample Parallel-CFG with scalar computations prior and after the parallel region.

6.1.1 Local Dominance

A node n in the Parallel-CFG is locally dominated by a node m, iff n is dominated by m under CFG semantics. Additionally, the parallel nesting depth τ (Definition 1) of n has to be at least as high as the one of m, thus $\tau(n) \geq \tau(m)$ and the smallest parallel region containing m has to contain n too.

Local dominance ensures that a dominated point/value was reached/computed by the current thread, thus is locally available. It can be used for systems with high communication costs between concurrently running parts (e.g., distributed machines) as it will prevent optimizations from introducing additional "scalar-communication". In the example in Figure 22 neither `%pre` nor `%post` can be moved into the parallel region blocks S or P if local dominance is used.

6.1.2 Global Dominance

A node n in the Parallel-CFG is globally dominated by a node m, iff n is is dominated by m under CFG semantics or, alternatively, ... TODO: find a short wording.

Global dominance ensures that a dominated point/value was reached/computed by any thread, thus it is generally available. It can be used for systems with low communication costs between concurrently running parts (e.g., a multi-core chip) as it will allow optimizations to introduce additional "scalar-communication", e.g., in order to increase parallelism. In the example in Figure 22 both `%pre` and `%post` can be moved into the parallel region blocks S or P if global dominance is used.

7 Runtime Lowering

To implement parallel execution, the parallel regions need to be lowered to runtime library calls. For this process it is common to extract parallel loops and tasks into their own functions. Additionally, it might be needed to communicate features like privatization and reductions to the runtime library. While the implementation is library specific, the identification of common features can be done in a general way by a runtime independent "LoweringInfo" pass.

7.1 Loop Outlining

Parallel loops are, depending on their shape, amenable to special loop outlining that utilizes dedicated runtime library calls. We believe this process to be suitable for pattern matching, e.g., checking for one `fork interi`**`or`** instruction on the single loop latch block. While parallel loops can have any form, it is unlikely that more than a few occur in practice. Thus, we propose to match common loop structures and default to task outlining as a general alternative.

7.2 Task Outlining

A non-trivial `fork` instruction in a parallel region θ will spawn one task for each successors. To implement parallel execution of these tasks they will be outlined, possibly keeping one task in the original Parallel-CFG to be executed natively by the master thread (see also the `master` annotation of the `fork` instruction in Section 2.1). The part of the Parallel-CFG that needs to be outlined for a task is the maximal connected sub graph that starts at the successor blocks of the `fork` and does not contain a `join` instruction of θ. These sub graphs can be determined in a single scan of the parallel region.

7.3 LoweringInfo Pass

The LoweringInfo pass offers runtime independent information that is commonly needed to implement parallel regions using parallel runtime libraries. Some but certainly not all use cases are described here. To ease reading we will omit parallel loops but only argue about general parallel tasks. However, for the actual implementation one might want to distinguish them.

7.3.1 Scalar Communication

If a scalar is used and defined, inside as well as outside of a parallel task it has to be communicated. To determine the set of all scalars that need to be communicated for a given parallel region, a single scan of the region is sufficient. If the definition of a scalar is outside a task but a use is inside, the scalar is communicated into the task. If the use is outside but the definition is inside a task, the scalar is communicated out of the task. Both communication directions are commonly realized using memory e.g., a "struct" that contains a field for

each communicated scalar. It is the responsibility of the runtime dependent lowering pass to create the communication code as it might include special memory transfer calls. However, the LoweringInfo pass can identify all scalars that are communicated into and respectively out of a parallel region.

7.3.2 Privatized Variables

Several languages (OpenMP, OpenCL, ...) have a concept of private memory locations for each thread executing in parallel. In Parallel-IR these locations are made explicit using **`alloca`** instructions as shown i.e. in Figure 6. For task parallelism no additional steps are required in the presence of privatized variables. After task outlining there is no difference between a privatization **`alloca`** and any other **`alloca`** inside the task, regardless if it was introduced before or after the runtime lowering. For loop outlining and the consequent use of dedicated runtime library calls for parallel loops the situation is different. It is necessary to at least identify such privatized variables. To do so, it suffices to check the parallel loop for **`alloca`** instructions that are used by an `fork interi`**`or`** but also another instruction contained in the to be outlined sub graph of the Parallel-CFG. To implement the privatization, the lowering pass can "ignore" the `fork interi`**`or`** use of the **`alloca`** and move it to a new block that will become the new entry of the to be outlined sub graph. This is only legal just prior to the outlining as otherwise the **`alloca`** could be moved again which thereby could introduce sharing and data races not present before.

7.3.3 Reduction Detection

8 Roadmap

The development of Parallel-IR is about finding a trade-off between a future-proof design and practical short-term considerations. On the conceptual side, Parallel-IR shall be generic, simple and expressive. We should make an effort to show theoretical soundness and keep long-term requirements in mind. Eventually, Parallel-IR should abstract fully from concrete parallel APIs, just like LLVM-IR makes language front-ends independent of back-ends in LLVM today.

However, there is a demand for optimizing concrete parallel programming APIs such as OpenMP, CUDA kernels and Thrust right now. Even if theoretical work addresses these APIs, Parallel-IR will only be adopted if we provide working prototypes early on in the process.

To this end, the roadmap plan is build around vertical prototypes: we will start with a practical OpenMP-to-OpenMP optimizing compiler. However, every design decision in the development shall comply with long-term design goals.

8.1 Milestones

Conceptual Soundness Show/Proof that stock LLVM-optimizations do not affect parallel semantics in Parallel-IR. Also show that Parallel-IR well-formdness superseeds parallel loop ids.

OpenMP to OpenMP Implement a proof-of-concept vertical prototype for an OpenMP-optimizing compiler based on Parallel-IR. This milestone comprises of a frontend and backend for translating between OpenMP and Parallel-IR. There must not be any OpenMP runtime calls in the IR code. Instead, the front-end shall lower every OpenMP idiom to Parallel-IR including the runtime API. The OpenMP back-end must be able to translate every Parallel-IR idiom back to the OpenMP runtime library.

Parallelism Optimizations Develop a set of generic parallelism optimizations for Parallel-IR. While at this point the implementation only supports OpenMP, the optimizations must not rely on OpenMP artifacts in the IR.

CUDA to CUDA Translate CUDA to Parallel-IR, optimize it and again generate CUDA kernels and runtime calls (this includes CUDA Thrust because Thrust itself is build on top of the CUDA runtime).

TBD

Long term: Full abstraction of parallelism

8.1.1 Conceptual Soundness

Parallel-IR needs a well defined parallel semantics. It is crucial that existing LLVM optimization interact well with Parallel-IR to ease the transition to Parallel-IR. Standard compiler optimizations that do not "understand" the parallel semantics of Parallel-IR should not be able to break it. To this end,

Parallel-IR either needs to adhere or re-define generic notions such as dominance, the dependence order and liveness of values. Parallel-IR should replace all existing extensions in todays LLVM-IR to indicate parallelism, such as LLVM metadata.

8.1.2 OpenMP to OpenMP

We chose OpenMP as the target for our first prototype for two reasons: Firstly, we expect that all OpenMP idioms naturally translate to Parallel-IR. Secondly, The OpenMP runtime library can be inlined completely into the IR. There will not be any OpenMP artifacts in the Parallel-IR that the first parallelism optimizations have to account for.

8.1.3 CUDA to CUDA

It shall be possible to optimize kernels in the context of the host program, including optimizations such as kernel fusion (see Figure Figure 16b). It will not be possible to fully eliminate the CUDA runtime API from the program. CUDA supports device selection, which has no generic counterpart in Parallel-IR.

8.2 Long term: Full abstraction of parallelism

Parallel-IR should become the natural target for fully hardware-agnostic parallel languages. The front-ends for these languages will not insert any runtime APIs. At this point, it will be possible to leave the implementation of parallelism entirely to the compiler. For example, in the same Parallel-IR program a parallel loop nest could be lowered to CUDA while another parallel region mighty be a good match for CPU multi-threading. However, the user should have some level of control over the parallelism back-ends the compiler can use.