Par_incr: A library for incremental computation with support for parallelism

Dipesh Kafle | Mentor: Vesa Karvonen

What is incremental computation?

What is incremental computation?

- Software feature which, whenever a piece of data changes, attempts to save time by only recomputing those outputs which depend on the changed data.

What is incremental computation?

- Software feature which, whenever a piece of data changes, attempts to save time by only recomputing those outputs which depend on the changed data.
- Example: Excel, GUIs (e.g. React), Build systems, etc.

What is incremental computation?

- Software feature which, whenever a piece of data changes, attempts to save time by only recomputing those outputs which depend on the changed data.
- Example: Excel, GUIs (e.g. React), Build systems, etc.
- A bit more detailed description at Introducing Incremental

Par_incr

Par_incr

- Based on Efficient Parallel Self-Adjusting Computation(sort of a follow up paper to Adaptive Functional Programming)

Par_incr

- Based on Efficient Parallel Self-Adjusting Computation(sort of a follow up paper to Adaptive Functional Programming)

Existing similar libraries

- current_incr
- Jane Street's incremental

Par_incr

- Based on Efficient Parallel Self-Adjusting Computation(sort of a follow up paper to Adaptive Functional Programming)

Existing similar libraries

- current_incr
- Jane Street's incremental

So what's different?

Par_incr

- Based on Efficient Parallel Self-Adjusting Computation(sort of a follow up paper to Adaptive Functional Programming)

Existing similar libraries

- current_incr
- Jane Street's incremental

So what's different?

- You can leverage freshly introduced parallelism in `Par_incr`

How do we use the library?

How do we use the library?

We need to know about a few types before that

How do we use the library?

We need to know about a few types before that

- `'a Par_incr.Var.t`: Handle to the input of some computation. Changing this should correctly propagate changes

How do we use the library?

We need to know about a few types before that

- `'a Par_incr.Var.t`: Handle to the input of some computation. Changing this should correctly propagate changes
- `'a Par_incr.t` : This is different from `Var.t`. You call `Var.watch` on a `Var.t` to get `Par_incr.t`. The library provides combinators that operate on this to make larger computations.

How do we use the library?

We need to know about a few types before that

- `'a Par_incr.Var.t`: Handle to the input of some computation. Changing this should correctly propagate changes
- `'a Par_incr.t` : This is different from `Var.t`. You call `Var.watch` on a `Var.t` to get `Par_incr.t`. The library provides combinators that operate on this to make larger computations.
- `'a Par_incr.computation`: When you run `'a Par_incr.t`, you'll obtain `'a computation`. This internally stores everything required to propagate changes.

How do we use the library?(continued)

```
1  module Cutoff : sig
2    type 'a t =
3    | Never
4    | Always
5    | Phys_equal
6    | Eq of ('a -> 'a -> bool)
7    | F of (oldval:'a -> newval:'a -> bool)
8
9    val attach : 'a t -> 'a incremental -> 'a incremental
10 end
```

How do we use the library?(continued)

```
module Var : sig
  type 'a t
  val create : ?cutoff:'a Cutoff.t -> ?to_s:('a -> string) -> 'a -> 'a t
  val set : 'a t -> 'a -> unit
  val value : 'a t -> 'a
  val watch : 'a t -> 'a incremental
  module Syntax : sig
    val ( := ) : 'a t -> 'a -> unit
    val ( ! ) : 'a t -> 'a
  end
end
```

How do we use the library? (continued)

```
type 'a t
type 'a computation
type executor = {
  run : 'a. (unit -> 'a) -> 'a;
  par_do : 'a 'b. (unit -> 'a) -> (unit -> 'b) -> 'a * 'b;
}
module Cutoff: sig ... end
module Var : sig ... end
val return : 'a -> 'a t
val map : ?cutoff:'b Cutoff.t -> fn:('a -> 'b) -> 'a t -> 'b t
val map2 :
  ?cutoff:'c Cutoff.t ->
  ?mode:[`Par | `Seq] -> fn:('a -> 'b -> 'c) -> 'a t -> 'b t -> 'c t
val combine : 'a t -> 'b t -> ('a * 'b) t
val bind : fn:('a -> 'b t) -> 'a t -> 'b t
val par : left:'a t -> right:'b t -> ('a * 'b) t
val delay : (unit -> 'a t) -> 'a t
val value : 'a computation -> 'a
val run : executor:executor -> 'a t -> 'a computation
val propagate : 'a computation -> unit
val destroy_comp : 'a computation -> unit
```

How do we use the library? (continued)

How do we use the library? (continued)

- Define `Var.t` with some value.

How do we use the library? (continued)

- Define `Var.t` with some value.
- Perform `Var.watch` operation on `Var.t` and change it to `incremental`(`Par_incr.t`).

How do we use the library? (continued)

- Define `Var.t` with some value.
- Perform `Var.watch` operation on `Var.t` and change it to `incremental`(`Par_incr.t`).
- Use different combinators provided by the library on the `incremental`s and make even bigger `incremental`s.

How do we use the library? (continued)

- Define `Var.t` with some value.
- Perform `Var.watch` operation on `Var.t` and change it to `incremental`(`Par_incr.t`).
- Use different combinators provided by the library on the `incremental`s and make even bigger `incremental`s.
- Obtain value of a certain `incremental` by running it.

How do we use the library? (continued)

- Define `Var.t` with some value.
- Perform `Var.watch` operation on `Var.t` and change it to `incremental`(`Par_incr.t`).
- Use different combinators provided by the library on the `incremental`s and make even bigger `incremental`s.
- Obtain value of a certain `incremental` by running it.
- Running an `'a incremental` returns a `'a computation`.

How do we use the library? (continued)

- Define `Var.t` with some value.
- Perform `Var.watch` operation on `Var.t` and change it to `incremental`(`Par_incr.t`).
- Use different combinators provided by the library on the `incremental`s and make even bigger `incremental`s.
- Obtain value of a certain `incremental` by running it.
- Running an `'a incremental` returns a `'a computation`.
- When we change a `Var.t`(done with `Var.set` operation), it marks all dependent computations dirty.

How do we use the library? (continued)

- Define `Var.t` with some value.
- Perform `Var.watch` operation on `Var.t` and change it to `incremental`(`Par_incr.t`).
- Use different combinators provided by the library on the `incremental`s and make even bigger `incremental`s.
- Obtain value of a certain `incremental` by running it.
- Running an `'a incremental` returns a `'a computation`.
- When we change a `Var.t`(done with `Var.set` operation), it marks all dependent computations dirty.
- Running `propagate` operation on a dirty `computation` updates its value efficiently.

How do we use the library? (continued)

- Define `Var.t` with some value.
- Perform `Var.watch` operation on `Var.t` and change it to `incremental`(`Par_incr.t`).
- Use different combinators provided by the library on the `incremental`s and make even bigger `incremental`s.
- Obtain value of a certain `incremental` by running it.
- Running an `'a incremental` returns a `'a computation`.
- When we change a `Var.t`(done with `Var.set` operation), it marks all dependent computations dirty.
- Running `propagate` operation on a dirty `computation` updates its value efficiently.
- Destroy (with `destroy_comp` operation) `computation` when its no more required.

Example

Sequential sum_range

```
1  let rec sum_range ~lo ~hi xs =
2    Par_incr.delay @@ fun () ->
3    if hi - lo <= 1 then begin
4      xs.(lo)
5    end
6    else
7      let mid = lo + ((hi - lo) asr 1) in
8      Debug.attach ~fn:Int.to_string
9        Par_incr.Syntax.(
10         let+ lhalf = sum_range ~lo
           ↪  ~hi:mid xs
11         and+ rhalf = sum_range ~lo:mid
           ↪  ~hi xs in
12         lhalf + rhalf)
```

Example

Sequential sum_range

```
let rec sum_range ~lo ~hi xs =
  Par_incr.delay @@ fun () ->
  if hi - lo <= 1 then begin
    xs.(lo)
  end
  else
    let mid = lo + ((hi - lo) asr 1) in
    Debug.attach ~fn:Int.to_string
      Par_incr.Syntax.(
        let+ lhalf = sum_range ~lo
        ↪  ~hi:mid xs
        and+ rhalf = sum_range ~lo:mid
        ↪  ~hi xs in
        lhalf + rhalf)
```

Parallel sum_range

```
let rec sum_range_par ~lo ~hi xs =
  Par_incr.delay @@ fun () ->
  if hi - lo <= 1 then begin
    xs.(lo)
  end
  else
    Debug.attach ~fn:Int.to_string
      Par_incr.Syntax.(
        let mid = lo + ((hi - lo) asr
        ↪  1) in
        let& lhalf = sum_range_par ~lo
        ↪  ~hi:mid xs
        and& rhalf = sum_range_par
        ↪  ~lo:mid ~hi xs in
        lhalf + rhalf)
```

What happens internally?

Say we have something like this,

```
let count = 3
let arr = Array.init count (( + ) 1) (* [1,2,3] *)
let var_arr = Array.map (Par_incr.Var.create ~to_s:Int.to_string) arr
let t_arr = Array.map Par_incr.Var.watch var_arr
let executor =
  Par_incr.
    {
      run = (fun f -> f ());
      par_do =
        (fun l r ->
          let lres = l () in
          (lres, r ()));
    }
let () =
  let seq_comp = Par_incr.run ~executor (sum_range ~lo:0 ~hi:count t_arr) in
  Par_incr.dump_tree "sum-range.d2" seq_comp;
  Var.set var_arr.(1) 10;
  Par_incr.dump_tree "sum-range-after-change.d2" seq_comp;
  Par_incr.propagate seq_comp;
  Par_incr.dump_tree "sum-range-after-prop.d2" seq_comp;
  Par_incr.destroy_comp seq_comp
```

What happens internally?(continued)

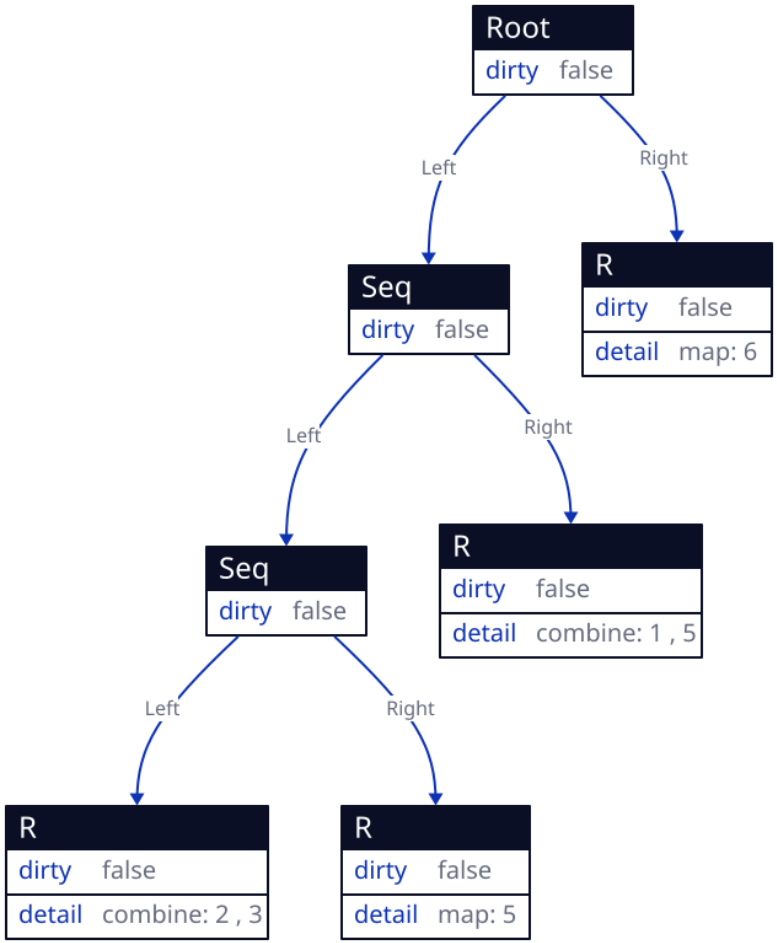

Figure 1: Internal representation of the computation

What happens internally?(continued)

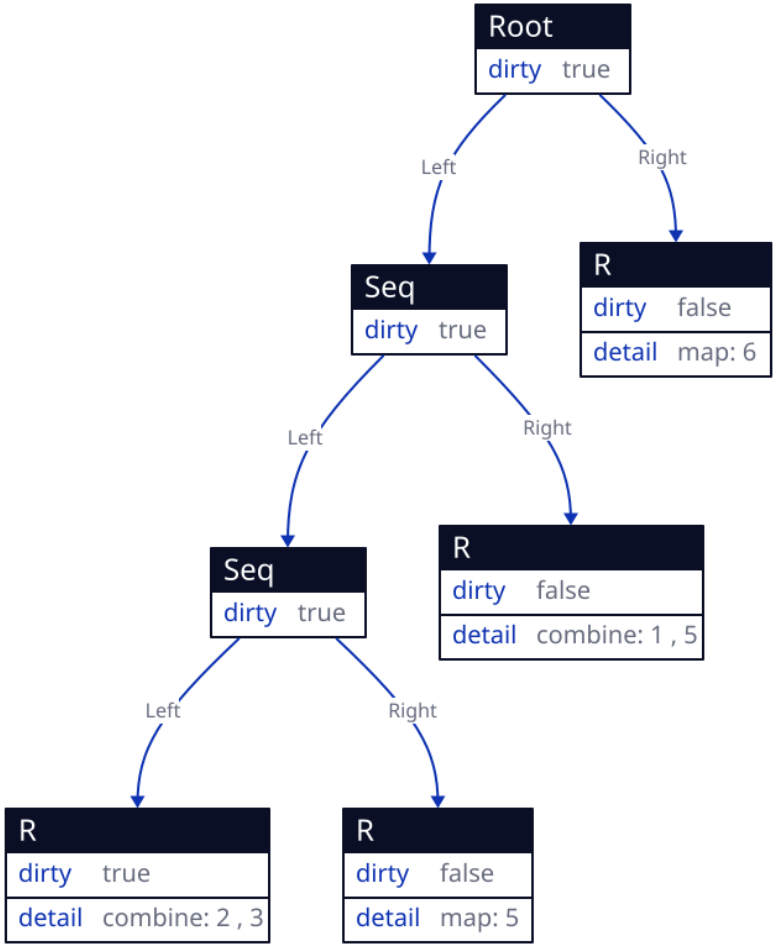

Figure 2: Computation after var_arr.(1) was changed

What happens internally?(continued)

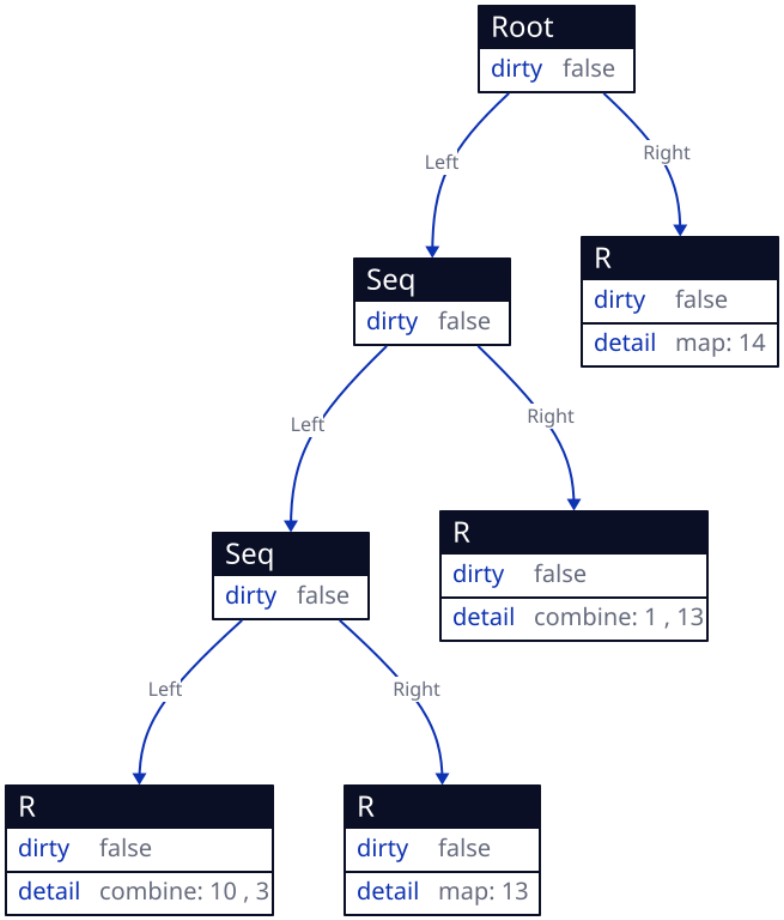

Figure 3: Computation after running propagate

What happens internally?(continue)

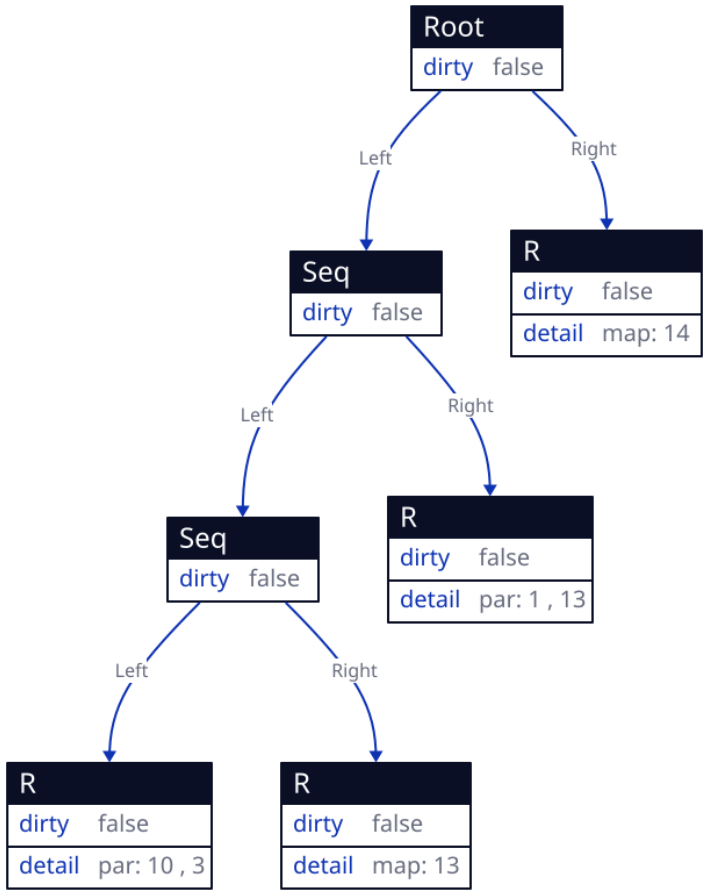

Figure 4: If we used sum_range_par instead

Performance

Performance

- Sequential performance within 2-4x of `Janestreet incremental` and pretty much always faster than `Current_incr`

Performance

- Sequential performance within 2-4x of `Janestreet incremental` and pretty much always faster than `Current_incr`
- Performance for parallelizable programs is almost always faster than `Janestreet incremental`. Sometimes by a lot and sometimes barely.

Performance

- Sequential performance within 2-4x of `Janestreet incremental` and pretty much always faster than `Current_incr`
- Performance for parallelizable programs is almost always faster than `Janestreet incremental`. Sometimes by a lot and sometimes barely.
- Link to benchmarks

Challenges

Challenges

- Diverting from the paper to a more functional style implementation

Challenges

- Diverting from the paper to a more functional style implementation
- Improving performance

Challenges

- Diverting from the paper to a more functional style implementation
- Improving performance
    - Having a standard documentation or a place to refer what optimizations kick in different cases would be super helpful.

Challenges

- Diverting from the paper to a more functional style implementation
- Improving performance
  - Having a standard documentation or a place to refer what optimizations kick in different cases would be super helpful.
  - Compiler could inline stuff across module to some extent, but couldn't inline some larger functions even when I wanted to(with the [@inlined] annotation). Something like this should be possible, and it can help immensely in optimization experiments.

Challenges

- Diverting from the paper to a more functional style implementation
- Improving performance
  - Having a standard documentation or a place to refer what optimizations kick in different cases would be super helpful.
  - Compiler could inline stuff across module to some extent, but couldn't inline some larger functions even when I wanted to(with the [@inlined] annotation). Something like this should be possible, and it can help immensely in optimization experiments.
- Profiling tools

Future Scope

Future Scope

- Library can probably be made faster, but it'll mostly require some major changes

Future Scope

- Library can probably be made faster, but it'll mostly require some major changes
- Implement some more programs using `Par_incr`

Future Scope

- Library can probably be made faster, but it'll mostly require some major changes
- Implement some more programs using `Par_incr`
- Implement the core of Differential Dataflow using `Par_incr`

THANK YOU