GRACL

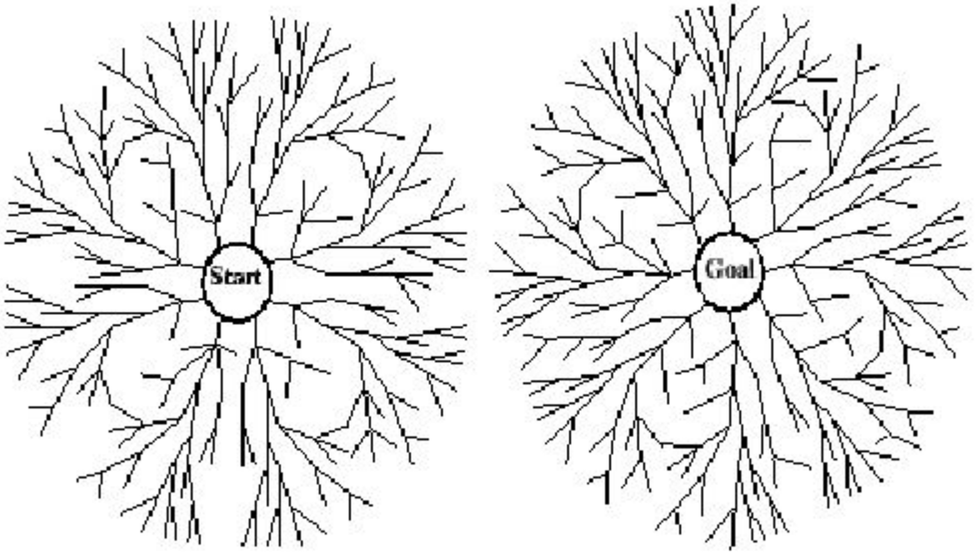
Defne Sonmez	dys2109	System Architect
Eilam Lehrman	esl2160	Language Guru
Hadley Callaway	hcc2134	Manager
Maya Venkatraman	mv2731	System Architect
Pelin Cetin	pc2807	Tester

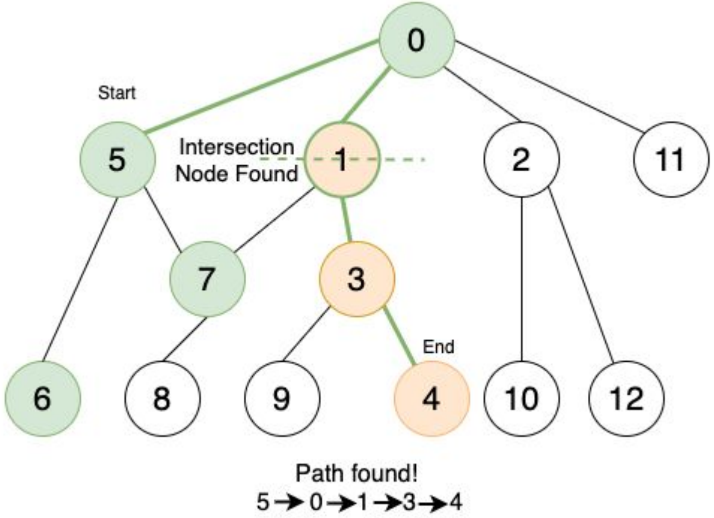
OVERVIEW

What is GRACL?

- GRAph Concurrency Language
- Enables common graph algorithms such as Depth-First-Search (DFS) and Dijkstra
- Leverages **concurrency** and built-in data structures to **initialize and modify graphs**
- Allows unique concurrent graph algorithms that may **converge more quickly** than their traditional counterparts
- Syntax with elements from Java, Python, and C
- Following features are available to the user:
  - Types: **Graph, Node, Edge, Nodelist, Edgelist, DoubleTable, IntTable**
  - Keywords **hatch** and **synch** for thread manipulation

Motivation

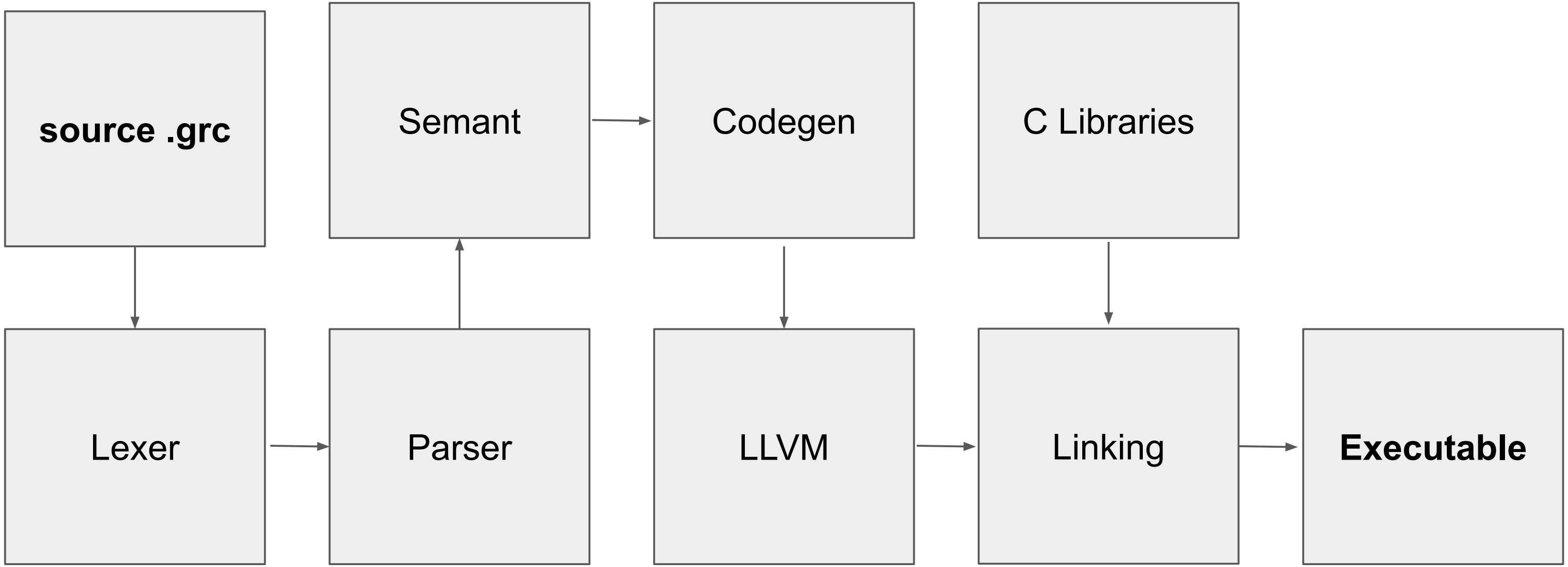

Start
Goal


Start
Intersection
Node Found
End
0
5
1
2
11
7
3
6
8
9
4
10
12
Path found!
5 → 0 → 1 → 3 → 4

Language Features

- Statically scoped
- Strongly and statically typed
- Pass by value
- Mutable data types
- Block scoping
- Imperative language
- All objects are on the heap

Compiler Architecture



Syntax

```
double example() {
    Graph g = createGraph(2);
    Node n1 = g.addNode(“hello”);
    Node n2 = g.addNode(“goodbye”);
    Edge e = g.createEdge(n1, n2, 10.0);
    return e.weight();
}
```

Hatch and Synch Syntax

```
hatch nodelist normalDFS_start(goal, myPath, path) {
    // code that parent thread executes before ending brace
}
synch lockedObject {
        // code performed while the implicit lock on lockedObject is held
}
```

COMPILER

Hatch in Codegen

```
| SHatch(nl, func, args, stmts) -> let argtypes = A.Node::(List.map (fun (t, _) -> t) args) in
(* Wrapper Struct *)
let hatch_t = L.named_struct_type context "hatch_args" in
let _       = L.struct_set_body hatch_t (Array.of_list (List.map ltype_of_typ argtypes)) false in

(* Unwrapper Function *)
let unwrapper_func_t = L.function_type string_t [| string_t |] in
let unwrapper_func = L.define_function ("hatch_unwrapper_" ^ func) unwrapper_func_t the_module in
let unwrap_builder = L.builder_at_end context (L.entry_block unwrapper_func) in
let (fdef, _) = StringMap.find func function_decls in let hatching_func_t = L.type_of fdef in

let void_alloca = L.build_alloca string_t "void_ptr" unwrap_builder in
let struct_alloca = L.build_alloca (L.pointer_type hatch_t) "wrapper" unwrap_builder in
let _ = L.build_store (L.param unwrapper_func 0) void_alloca unwrap_builder in

let load_ptr = L.build_load void_alloca "void_ptr" unwrap_builder in
let ptr_cast = L.build_bitcast load_ptr (L.pointer_type hatch_t) "cast_ptr" unwrap_builder in
let _ = L.build_store ptr_cast struct_alloca unwrap_builder in

let init_arg i =
  let struct_load = L.build_load struct_alloca "struct_ptr" unwrap_builder in
  let arg_gep = L.build_in_bounds_gep struct_load [|L.const_int i32_t 0; L.const_int i32_t i|] "arg_gep" unwrap_builder in
  let arg_load = L.build_load arg_gep "arg" unwrap_builder in arg_load
in
let func_args = Array.init (Array.length (L.params fdef)) (init_arg) in
let _ = L.build_call fdef func_args (if L.return_type (L.return_type hatching_func_t) = void_t then "" else "result") unwrap_builder in

let _ = L.build_ret (L.const_null string_t) unwrap_builder in
```

Get types to create struct dynamically

Constructs arguments to call user function by unwrapping the argument struct

```
(* Hatch Threads *)
let length_alloca = L.build_alloca i32_t "nl_length" builder in
let list_length = expr builder (A.Int, SCall("length_NL", [nl])) in
let _ = L.build_store list_length length_alloca builder in

let pthreads_alloca = L.build_alloca (L.pointer_type i64_t) "pthread_array" builder in
let args_alloca = L.build_alloca (L.pointer_type hatch_t) "args_array" builder in
let malloc_func_t = L.function_type string_t [| i64_t |] in
let malloc_func = L.declare_function "malloc" malloc_func_t the_module in

let build_malloc numval typ =
  let length_load = L.build_load length_alloca "length" builder in
  let sext = L.build_sext length_load i64_t "sext_length" builder in
  let mul = L.build_mul numval sext "bytes" builder in
  let malloc = L.build_call malloc_func [| mul |] "malloc" builder (*L.build_array_malloc string_t mul "malloc" builder*) in
  let bitcast = L.build_bitcast malloc typ "cast_mem" builder in
bitcast in
let _ = L.build_store (build_malloc (L.const_int i64_t 8) (L.pointer_type i64_t)) pthreads_alloca builder in
let _ = L.build_store (build_malloc (L.const_int i64_t (List.fold_left (fun i t -> i + size_of_typ t) 0 argtypes))
        (L.pointer_type hatch_t)) args_alloca builder in

let i_alloca = L.build_alloca i32_t "i" builder in
let _ = L.build_store (L.const_int i32_t 0) i_alloca builder in
```

Calculates size of struct

```
(* For Loop *)
let list_alloca = L.build_alloca nodelist_pointer "list" builder in
  let item_alloca = L.build_alloca nodelistitem_pointer "item" builder and
  _ = L.build_store (expr builder nl) list_alloca builder in
  let list_load = L.build_load list_alloca "list" builder in
  let list_gep = L.build_in_bounds_gep list_load [|L.const_int i32_t 0; L.const_int i32_t 1|] "list_gep" builder in
  let list_pointer = L.build_load list_gep "item_ptr" builder in
  let _ = L.build_store list_pointer item_alloca builder in


  let pred_bb = L.append_block context "hatch_for" the_function in
  ignore(L.build_br pred_bb builder);


  let body_bb = L.append_block context "hatch_for_body" the_function in
  let endfor_bb = L.append_block context "hatch_end_for" the_function in
  let body_builder = L.builder_at_end context body_bb in
  let item_load = L.build_load item_alloca "item" body_builder in
  let item_gep = L.build_in_bounds_gep item_load [|L.const_int i32_t 0; L.const_int i32_t 0|] "item_gep" body_builder in
  let set_up_struct ld num =
    let load_struct = L.build_load args_alloca "arg" body_builder in
    let load_i = L.build_load i_alloca "i" body_builder in
    let sext = L.build_sext load_i i64_t "i" body_builder in
    let gep1 = L.build_in_bounds_gep load_struct [| sext |] "gep1" body_builder in
    let field_gep = L.build_in_bounds_gep gep1 [| L.const_int i32_t 0; L.const_int i32_t num |] "field_gep" body_builder in
    let _ = L.build_store ld field_gep body_builder in ()
  in
  let _ = set_up_struct (L.build_load item_gep "val" body_builder) 0 in
  let _ = Array.init (Array.length (L.params fdef) - 1) (fun x -> set_up_struct (expr body_builder (List.nth args x)) (x + 1)) in
```

Fill each struct with the arguments

Block Scoping

```
  Block sl -> let st = StringMap.empty::st in
    let rec check_stmt_list st = function
        [Return _ as s] -> [check_stmt st s]
      | Return _ :: _   -> raise (Failure "nothing may follow a return")
      | Block sl :: ss  -> check_stmt_list (StringMap.empty::st) (sl @ ss) (* Flatten blocks *)
      | LoclBind(b) as lb :: ss -> let add_local typ name = if StringMap.mem name (List.hd st) then raise (Failure ("Cannot redeclare " ^ name))
        else StringMap.add name (typ, "var" ^ string_of_int (count.(0)) ^ "_" ^ name) (List.hd st) and (t,n) = strip_val b and _ = count.(0) <- 1 + count.(0) in
        let updated_table = (add_local t n)::(List.tl st) in
        let stm = check_stmt updated_table lb in stm :: check_stmt_list updated_table ss
      | BlockEnd :: ss -> SBlockEnd::check_stmt_list (List.tl st) ss
      | s :: ss         -> let stm = check_stmt st s in stm :: check_stmt_list st ss (* stm is VERY important here *)
      | []              -> []
    in SBlock(check_stmt_list st sl)

  LoclBind(b) ->
 egin match b with
  | Dec(t,n) -> if t = Void then raise (Failure ("illegal void local " ^ n)) else
    let (_, newname) = type_of_identifier n st in StringHash.replace locals newname (SDec(t,newname)); SExpr(t, SId newname)
  | Decinit(t,n,e) as di ->
  if t = Void then raise (Failure ("illegal void local " ^ n)) else
  let (rt, ex) = expr st e in
    let err = "illegal assignment " ^ string_of_typ t ^ " = " ^
      string_of_typ rt ^ " in " ^ string_of_vdecl di
    in let _ = check_assign t rt err and (_, newname) = type_of_identifier n st
      in StringHash.replace locals newname (SDecinit(t, newname, (rt, ex))); SExpr(t, SAssign(newname, (rt, ex))) end
```

Alpha renaming

Pop top symbol table thanks to special expression

Track locals to allocate space at top of function

C BACKEND

```c
struct Node {
    pthread_mutex_t lock;
    int id; // Only used under the hood
    char *data;
    bool visited;
    struct EdgeList* edges;
    struct Node* precursor;
    double cost;
    int parent_graph_id;
    bool deleted;
};

struct Edge {
    pthread_mutex_t lock;
    double weight;
    struct Node* start;
    struct Node* end;
    bool deleted;
};

struct EdgeListItem {
    struct Edge* edge;
    struct EdgeListItem* next;
    struct EdgeListItem* prev;
};

struct EdgeList {
    pthread_mutex_t lock;
    struct EdgeListItem* head;
    struct EdgeListItem* tail;
};
struct NodeListItem {
    struct Node* node;
    struct NodeListItem* next;
    struct NodeListItem* prev;
};

struct NodeList {
    pthread_mutex_t lock;
    struct NodeListItem* head;
    struct NodeListItem* tail;
};

struct DataItem {
    struct Node* key;
    struct EdgeList* value;
};

struct Graph
{
    struct DataItem* hashArray;
    struct NodeList* nodes;
    int size;
    int id_num;
    int graph_id_local;
    int occupied;
};

struct IntTableItem {
    struct IntTableItem* next;
    struct IntTableLLItem* entry;
};

struct IntTableLLItem {
    struct Node* key;
    int value;
};

struct DoubleTableItem {
    struct DoubleTableItem* next;
    struct DoubleTableLLItem* entry;
};

struct DoubleTableLLItem {
    struct Node* key;
    double dub;
};

struct IntTable
{
    pthread_mutex_t lock;
    struct IntTableItem* arr;
    struct NodeList* keys;
    int size;
    int graph_id;
};

struct DoubleTable
{
    pthread_mutex_t lock;
    struct DoubleTableItem* arr;
    struct NodeList* keys;
    int size;
    int doubleId;
    int graph_id;
};
```

Node Type

- Notable that we tried to build a rich data type for node to give the user **many options**
    - Precursor vs maintaining a nodelist
    - Cost vs IntTable or DoubleTable (as traditionally used in Dijkstra)
- Support a large number of graph algorithms

```c
struct Node {
    pthread_mutex_t lock;
    int id; // Only used under the hood
    char *data;
    bool visited;
    struct EdgeList* edges;
    struct Node* precursor;
    double cost;
    int parent_graph_id;
    bool deleted;
};
```

Graph Structure

- **hashArray** is an array of **node keys** mapped to **EdgeList values**
- In the values we store all edges that point to the node in the key
- This makes **removal O(1)**
- Addition of nodes/edges **O(1)** as well

```
struct Graph
{
    struct DataItem* hashArray;
    struct NodeList* nodes;
    int size;
    int id_num;
    int graph_id_local;
    int occupied;
};
```

Collision Handling

- IntTable and DoubleTable are implemented with a different form of collision handling than the underlying Graph type
- For both, users may input a **predicted size** of items they think they will input
- However, we handle the user **exceeding their original bound**

In **IntTable/DoubleTable**:

Type is user exposed, and we don't internally id the nodes added; we implement a true form of hashing where **buckets created at every hash index**

The user may have worse **operation performance** if they exceed their expected size

In **Graph** object:

Type isn't user exposed and we **internally id nodes**

We use an array that we double if original size exceeded; **O(1)** access guaranteed

Lazy Delete

- Instead of freeing Node or Edge objects when the user removes them, **we mark the deleted boolean for lazy deletion**
- For Node and Edge accessor functions and some Graph functions, **we check if the object has been deleted first**
- Deliberate choice for **defined, clear behavior**

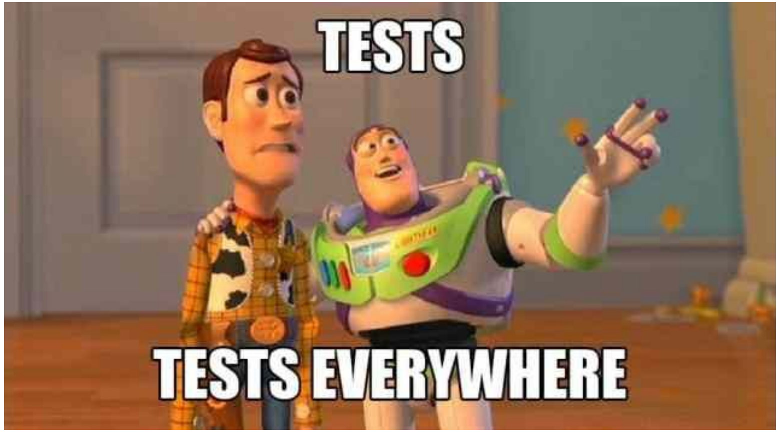
```c
/* Returns a boolean representing
 * if the node has already been visited. */
bool visited(struct Node* node)
{
    if (node->deleted) {
        fprintf(stderr, "visited: Node deleted\n");
        exit(1);
    }
    return node->visited;
}
```

TESTING

Testing Suite

Three testing scripts

- **./testall.sh** for all the GRACL files
    - 119 tests
- **./test-script.sh** for all the C testing
    - 11 tests but tests are lengthier and more comprehensive
- **./time-dfs.sh** to get the time difference between concurrent DFS and normal DFS
- **valgrind** in Docker image, tests run to check for memory errors (not leaks)



./time-dfs.sh & Performance

```
[WARNING] Running as root is not recommended
A
C
F
J
T
W
[WARNING] Running as root is not recommended
A
C
F
J
T
W
[WARNING] Running as root is not recommended
A
C
F
J
T
W
Ran the tests ten times. Here are the results:
Time taken to run normaldfs ten times is: 144628 milliseconds
Time taken to run concdfs ten times is: 135506 milliseconds
On average, normaldfs ran in: 14462 milliseconds
On average, concdfs ran in: 13550 milliseconds
root@5fe8937a2727:/home/microc#
root@5fe8937a2727:/home/microc#
```

Demos

- Dijkstra
- Non-concurrent vs. Concurrent DFS (takes ~5 min)
- Implementation and performance
- Bidirectional Search

Future Work

- Concurrent Tarjan’s algorithm
- Multiple returns
- More polymorphism
- Int/DoubleTable taking expressions, not just IDs
- Easier large graph creation
- Memory
    - Freeing nodes and edges after removal rather than lazy delete

“If I held a gun against your head, would you be able to write concurrency in C?”
- Stephen Edwards, 2021