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
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

- **source .grc**
- **Lexer**
- **Semant**
- **Parser**
- **Codegen**
- **LLVM**
- **C Libraries**
- **Linking**
- **Executable**
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

```
436 | SHash(nl, func, args, stmts) -> let argtypes = A.Node::(List.map (fun (t, _) -> t) args) in
437 | (* Wrapper Struct *)
438 | let hatch_t = L.named_struct_type context "hatch_args" in
439 | let _ = L.struct_set_body hatch_t (Array.of_list (List.map ltype_of_typ argtypes)) false in
440 | (* Unwrapper Function *)
441 | let unwrapper_func_t = L.function_type_string_t [] [] string_t [] in
442 | let unwrapper_func = L.define_function ("hatch_unwrapper" ^ func) unwrapper_func_t the_module in
443 | let unwrap_builder = L.builder_at_end context (L.entry_block unwrapper_func) in
444 | let (fdef, _) = StringMap.find func function_decls in let hatching_func_t = L.type_of fdef in
445 |
446 | let voidalloca = L.build_alloca string_t "void_ptr" unwrap_builder in
447 | let structalloca = L.build_alloca (L.pointer_type hatch_t) "wrapper" unwrap_builder in
448 | let _ = L.build_store (L.param unwrapper_func 0) voidalloca unwrap_builder in
449 |
450 | let load_ptr = L.build_load voidalloca "void_ptr" unwrap_builder in
451 | let ptr_cast = L.build_bitcast load_ptr (L.pointer_type hatch_t) "cast_ptr" unwrap_builder in
452 | let _ = L.build_store ptr_cast structalloca unwrap_builder in
453 |
454 | let init_arg i =
455 |   let struct_load = L.build_load structalloca "struct_ptr" unwrap_builder in
456 |   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
457 |   let arg_load = L.build_load arg_gep "arg" unwrap_builder in arg_load
458 |
459 | let func_args = Array.init (Array.length (L.params fdef)) (init_arg i) in
460 | 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
461 |
462 | let _ = L.build_ret (L.const_null string_t) unwrap_builder in
```
let lengthalloca = 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 [i 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
let listalloca = L.buildalloca nodelist_pointer "list" builder in
let itemalloca = L.buildalloca nodelistitem_pointer "item" builder and
_ = L.build_store (expr builder n1) listalloca builder in
let list_load = L.build_load listalloca "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 itemalloca builder in

let pred_bb = L.append_block context "hatch_for" the_function in
generate(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 itemalloca "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_structure = L.build_load argsalloca "arg" body_builder in
  let load_i = L.build_load ialloca "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_structure [[ 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))
Block Scoping

```ocaml
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) in
                                   let updated_table = (add_local t n)::(List.tl st) in
                                   let stmt = check_stmt updated_table lb in stmt :: check_stmt_list updated_table ss
        | BlockEnd :: ss -> SBlockEnd::check_stmt_list (List.tl st) ss
        | _ :: ss -> let stmt = check_stmt st s in stmt :: check_stmt_list st ss (* stmt is VERY important here *)
        | [] -> []
in SBlock(check_stmt_list st sl)
```

- **Alpha renaming**: Twisting the names of local variables to ensure they are unique.
- **Pop top symbol table thanks to special expression**: Using a special expression to pop the top of the symbol table.
- **Track locals to allocate space at top of function**: Tracking local variables to allocate space at the top of the function.
C BACKEND
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 NodeList {
    pthread_mutex_t lock;
    struct NodeListItem* next;
    struct NodeListItem* prev;
};

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 id_num;
    int graph_id_local;
    int occupied;
};

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

struct DoubleTableLLItem {
    struct DoubleTableItem* entry;
    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 DoubleTableLLItem* entry;
    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

```c
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

[WARNING] Running as root is not recommended

[WARNING] Running as root is not recommended

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#
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