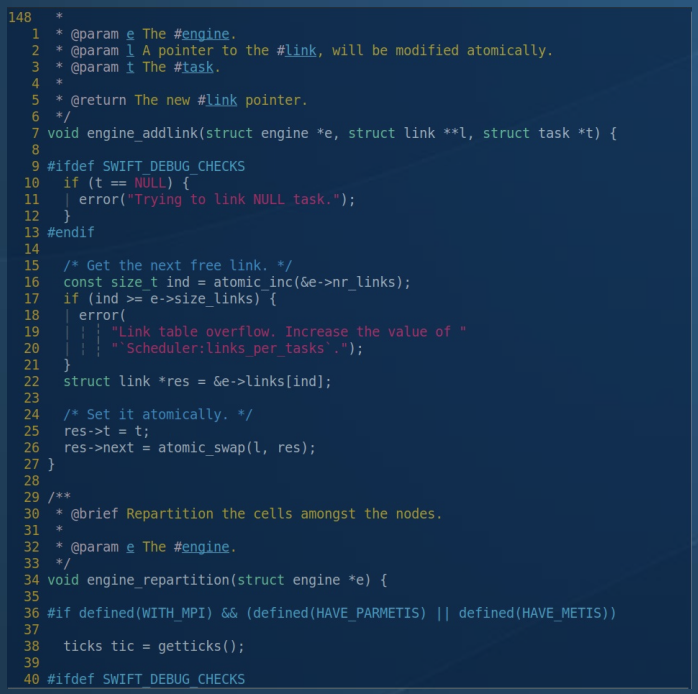
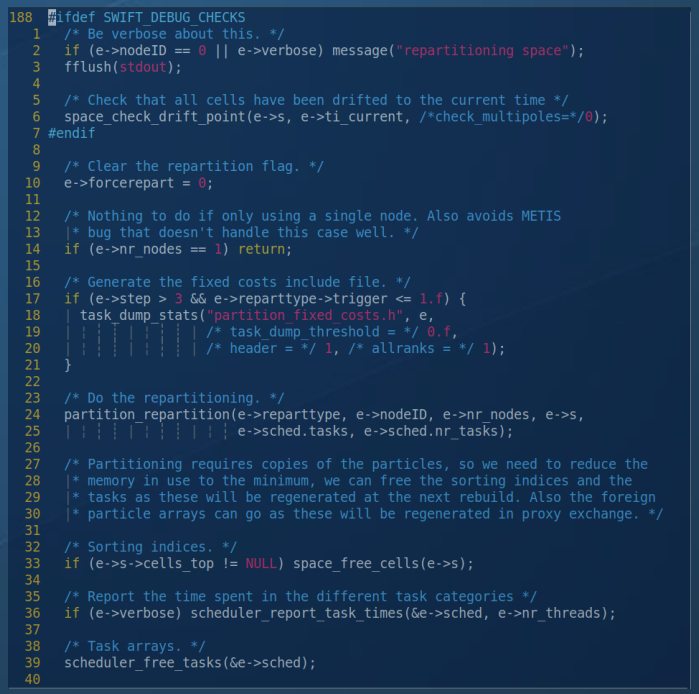
Code Development: Best Practices

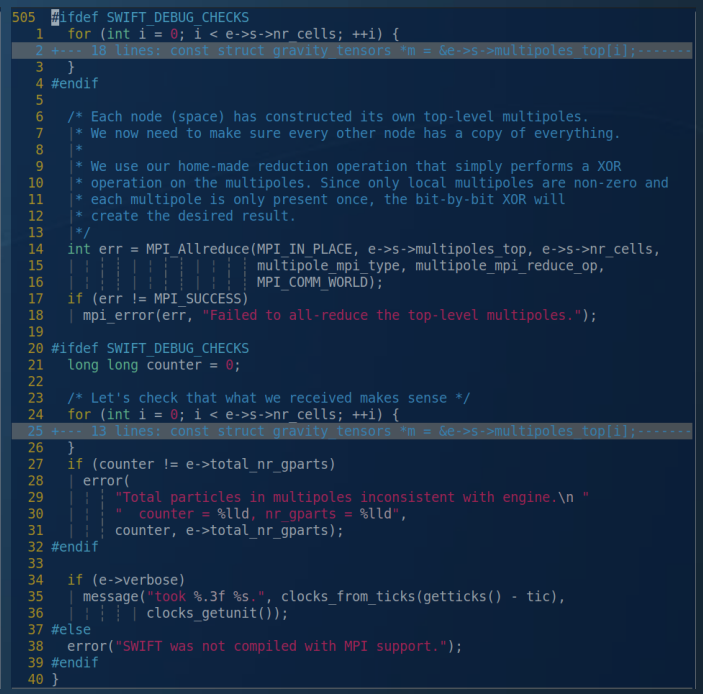
Generally and in SWIFT



```
148  *
 1   * @param e The #engine.
 2   * @param l A pointer to the #link, will be modified atomically.
 3   * @param t The #task.
 4   *
 5   * @return The new #link pointer.
 6   */
 7  void engine_addlink(struct engine *e, struct link **l, struct task *t) {
 8
 9  #ifdef SWIFT_DEBUG_CHECKS
10    if (t == NULL) {
11      error("Trying to link NULL task.");
12    }
13  #endif
14
15    /* Get the next free link. */
16    const size_t ind = atomic_inc(&e->nr_links);
17    if (ind >= e->size_links) {
18      error(
19          "Link table overflow. Increase the value of "
20          "`Scheduler:links_per_tasks`.");
21    }
22    struct link *res = &e->links[ind];
23
24    /* Set it atomically. */
25    res->t = t;
26    res->next = atomic_swap(l, res);
27  }
28
29  /**
30   * @brief Repartition the cells amongst the nodes.
31   *
32   * @param e The #engine.
33   */
34  void engine_repartition(struct engine *e) {
35
36  #if defined(WITH_MPI) && (defined(HAVE_PARMETIS) || defined(HAVE_METIS))
37
38    ticks tic = getticks();
39
40  #ifdef SWIFT_DEBUG_CHECKS
```



```
188  #ifdef SWIFT_DEBUG_CHECKS
 1     /* Be verbose about this. */
 2     if (e->nodeID == 0 || e->verbose) message("repartitioning space");
 3     fflush(stdout);
 4
 5     /* Check that all cells have been drifted to the current time */
 6     space_check_drift_point(e->s, e->ti_current, /*check_multipoles=*/0);
 7  #endif
 8
 9     /* Clear the repartition flag. */
10     e->forcerepart = 0;
11
12     /* Nothing to do if only using a single node. Also avoids METIS
13      * bug that doesn't handle this case well. */
14     if (e->nr_nodes == 1) return;
15
16     /* Generate the fixed costs include file. */
17     if (e->step > 3 && e->reparttype->trigger <= 1.f) {
18       task_dump_stats("partition_fixed_costs.h", e,
19                       /* task_dump_threshold = */ 0.f,
20                       /* header = */ 1, /* allranks = */ 1);
21     }
22
23     /* Do the repartitioning. */
24     partition_repartition(e->reparttype, e->nodeID, e->nr_nodes, e->s,
25                           e->sched.tasks, e->sched.nr_tasks);
26
27     /* Partitioning requires copies of the particles, so we need to reduce the
28      * memory in use to the minimum, we can free the sorting indices and the
29      * tasks as these will be regenerated at the next rebuild. Also the foreign
30      * particle arrays can go as these will be regenerated in proxy exchange. */
31
32     /* Sorting indices. */
33     if (e->s->cells_top != NULL) space_free_cells(e->s);
34
35     /* Report the time spent in the different task categories */
36     if (e->verbose) scheduler_report_task_times(&e->sched, e->nr_threads);
37
38     /* Task arrays. */
39     scheduler_free_tasks(&e->sched);
40
```



```
505  #ifdef SWIFT_DEBUG_CHECKS
 1     for (int i = 0; i < e->s->nr_cells; ++i) {
 2  +--- 18 lines: const struct gravity_tensors *m = &e->s->multipoles_top[i];------
 3     }
 4  #endif
 5
 6     /* Each node (space) has constructed its own top-level multipoles.
 7      * We now need to make sure every other node has a copy of everything.
 8      *
 9      * We use our home-made reduction operation that simply performs a XOR
10      * operation on the multipoles. Since only local multipoles are non-zero and
11      * each multipole is only present once, the bit-by-bit XOR will
12      * create the desired result.
13      */
14     int err = MPI_Allreduce(MPI_IN_PLACE, e->s->multipoles_top, e->s->nr_cells,
15                             multipole_mpi_type, multipole_mpi_reduce_op,
16                             MPI_COMM_WORLD);
17     if (err != MPI_SUCCESS)
18       mpi_error(err, "Failed to all-reduce the top-level multipoles.");
19
20  #ifdef SWIFT_DEBUG_CHECKS
21     long long counter = 0;
22
23     /* Let's check that what we received makes sense */
24     for (int i = 0; i < e->s->nr_cells; ++i) {
25  +--- 13 lines: const struct gravity_tensors *m = &e->s->multipoles_top[i];------
26     }
27     if (counter != e->total_nr_gparts)
28       error(
29           "Total particles in multipoles inconsistent with engine.\n "
30           "  counter = %lld, nr_gparts = %lld",
31           counter, e->total_nr_gparts);
32  #endif
33
34     if (e->verbose)
35       message("took %.3f %s.", clocks_from_ticks(getticks() - tic),
36               clocks_getunit());
37  #else
38     error("SWIFT was not compiled with MPI support.");
39  #endif
40  }
```

Mladen Ivkovic
Durham University

Leiden, 19. September 2023

Note

- These slides contain only the part of my talk relating to the gdb/valgrind live demo.
- The state of the demos as presented then are stored in the git repository in the branch `freeze-swiftcon-2023-09`
- full slides of the workshop are available on
  - https://1drv.ms/b/s!Aq715l3GOLnojmRg-YWc45UN9m0z?e=vabJ3l
  - https://mladenivkovic.github.io/work.html

External Debugging Tools

- gdb
  - tracebacks, inspection
- valgrind
  - memory leaks

- Live demo scripts
  - https://github.com/mladenivkovic/debugging-essentials-demo

gdb

- Compile your program with debug symbols -g
- $ gdb /path/to/executable
- $ gdb **--args** path/to/executable **--arg1 --arg2**
- Useful commands:
  - b (break) : set a break point
  - l (list): show source code in CLI
  - p (print): print variables
    - For pointers: p *pointer works too!
  - bt (backtrace): show stack trace
  - c (continue): continue run
  - pi (python-interactive): open a python interpreter

gdb + core dumps

- It’s possible to enable core dumps:
  - when program encounters error, write down what is currently in memory instead of just quitting
  - Enabling core dumps on linux:
    - $ ulimit -S -c unlimited
    - Note: your sysadmins may have disabled this.
    - Default core dump location may vary. On HPC systems, it’s often set to workdir. On Ubuntu 21+, var/lib/apport/coredump/
- gdb can read that back in and allow you to debug!
- $ gdb -c core.XXXX path/to/executable

gdb – some tricks

- „value has been optimized out” workaround
  - Compile entire program without optimization
  - … or tell compiler not to optimize specific function you’re looking at:

```
#pragma GCC push_options
#pragma GCC optimize ("O0")
your code
#pragma GCC pop_options
```

gdb + MPI

- $ mpirun -n 4 xterm -e gdb -ex run \
  --args ../../swift_mpi --arg1 --arg2
  - Launch 4 MPI ranks
  - Launch a terminal (xterm) and execute subsequent command (-e)
  - Launch gdb and immediately execute „run" command
  - gdb flag: The executable (swift_mpi) will need command line arguments (--arg1, --arg2)
  - Launch swift
- *MPI_Abort*() may exit gracefully instead of raising/signalling an error.
  - Simplest workaround: Replace with *abort()* while debugging

valgrind

- Track down memory leaks
- $ valgrind path/to/your/executable
- $ valgrind –leak-check=full path/to/executable
  - For more details, traces, etc