2 Work-Stealing for Task Trees — extended[1]

Work stealing is a popular efficient technique for performing load balancing in multicore computations. In traditional schemes, the work-stealing is *receiver-initiated*: workers that run out of work are responsible for *stealing* tasks. In a dual approach, called *sender-initiated work-stealing*, workers with tasks are responsible for actively sharing their tasks with workers that are out of work.

In this challenge we investigate work-stealing in the context of a *binary tree of tasks*. Each task has at most two child *subtasks* and a task may be executed only *after* their parent task; the root task must be executed first, therefore. The order in which tasks can be executed is otherwise not restricted.

Let $P > 0$ be the number of workers; each worker has a unique ID `i` in the range 0 to $P-1$. Each worker has a double-ended queue `q[i]` representing the tasks currently ready to execute and assigned to this worker; these start off empty, and the root task is then assigned to worker 0. Our algorithms are each built around the same key main function, shown in pseudo code below:

```
typedef int task        // tasks are represented by their IDs
const int nTasks        // number of tasks (IDs 0,1,...,nTasks-1)
const task NO_TASK = -1 // special code to denote ‘no child task’
const task ROOT_TASK = 0 // ID of the root task
task subtask[nTasks][2] // maps task ID to child task IDs / NO_TASK
bool executed[nTasks]  // marks executed tasks; initially 0 (false)

const int P       // number of workers
deque<task> q[P] // double-ended queue per worker; initially empty

// entry point for each worker (calling this concurrently)
void main(int i) // i = ID of the worker; 0 for the initial worker
  if i = 0 // the worker who starts things off
    push_bottom(q[i], ROOT_TASK)
  repeat // until termination
    if (empty(q[i])) // if out of work, try to acquire a task
      acquire(i)     // scheme-dependent function; see later
    else // pick a task and execute it
      task t = pop_bottom(q[i])
      communicate(i) // scheme-dependent function; see later
      execute(i, t)

// execution of a task t by worker i
void execute(int i, task t)
  // perform some task-specific computation; omitted for simplicity
  executed[t] += 1 // flag the task as executed
  // then schedule the subtasks
  add_task(i, subtask[t][1])
  add_task(i, subtask[t][0])

// called for scheduling a task t into worker i’s queue
void add_task(int i, task t)
  if t != NO_TASK
    push_bottom(q[i], t)
```

[1]We warmly thank Arthur Charguéraud for contributing the idea for this challenge.

The array `subtask` (which is never mutated in the code; you may assume this to be immutable if it helps you) expresses the tree structure of tasks: looking up a task's ID in the array gives an array with two elements, storing the task IDs of its respective subtasks (or the special value `NO_TASK`).

We assume an existing suitable implementation of a double-ended queue (the `deque` type in our pseudo code). You do *not* need to implement this type for these challenges. However, since the code we are concerned with interacts with these queues, you will need specifications for five functions on these queues:

`empty(Q)` returning a boolean indicating whether the queue `Q` is empty.

`peek_top(Q)` returning the element at the start of `Q` (without removing it).

`pop_top(Q)` removing the top element from `Q` and returning it.

`push_bottom(Q,T)` which modifies the queue `Q`, adding the task `T` at the end.

`pop_bottom(Q)` removing the bottom/end element from `Q` and returning it.

The variation between different task-handling schemes is expressed by changing (only) the implementations of the `acquire` and `communicate` functions.

Version 0: Sequential task processing We start with a sequential scheme: here, we can assume `P` = 1 and so there is a unique worker executing the `main` function with ID (`i` parameter) 0. In this version, the two functions `acquire` and `communicate` are no-ops: their implementations are empty, and calling them does nothing. The (only) worker will initially add the root task to its queue, and continually execute a task in its queue, queueing up its subtasks, and so on. We do not handle *worker* termination in the code (which is complex for concurrent schemes), but all *tasks* should eventually be executed this way.

Tasks for version 0

(a) Formalise the assumption that the initial values stored in the array (of length two arrays) `subtask` define a valid binary tree rooted at task ID 0.

(b) Define suitable specifications for the queue functions listed above.

(c) Verify that the pseudocode functions given are memory-safe / crash free (assuming that the queue functions are similarly safe): in particular, verify that all array accesses performed are guaranteed to be within bounds.

(d) Verify that every task is executed at most once.

(e) Verify that all task dependencies (as expressed by the tree structure) are respected: a subtask is never executed before its parent.

(f) Verify that all tasks are eventually executed.

(g) Verify that (after `ROOT_TASK` has been inserted by worker 0) all the worker queues (in version 0, the single queue `q[0]`) eventually become empty.

Version 1: Sender-initiated Work-stealing The following alternative implementations of the `acquire` and `communicate` functions (along with additional definitions/state as shown) implement a *sender-initiated work-stealing* scheme.

```
// extra definitions/state
const task WAITING = -2 // code for 'a task would be welcome'
const task NOT_WAITING = -3 // code for 'not receiving tasks'
task s[P]; // communication cells, all initially NOT_WAITING

// called by workers when running out of work
void acquire(int i)
  s[i] = WAITING           // 'a task would be welcome'
  while (s[i] == WAITING) // block until receiving a task
    noop
  add_task(i, s[i])
  s[i] = NOT_WAITING // technically optional

// consider pushing a task to an idle (different) worker
void communicate(int i)
  if (empty(q[i])) // cannot provide a task if we have none
    return
  int j = random in {0, ..., P-1}\{i} // pick a random other worker
  if s[j] != WAITING // check if that worker is waiting for tasks
    return           // give up if we picked a worker not waiting
  task t = peek_top(q[i])
  // attempt to atomically take the target communication slot
  bool r = compare_and_swap(&s[j], WAITING, t)
  if (r)          // we successfully wrote the task ID to s[j]
    pop_top(q[i]) // remove from our queue; now worker j gets it
```

The `acquire` function is only called when a worker has no tasks, and sets the worker's communication cell to `WAITING` to signal that a task can be assigned to it. It then busy-waits until this cell's value has been changed (to a task ID), and it then inserts this task and continues.

The `communicate` function represents worker `i` considering sending a task to another worker. If it has a task to send, it randomly guesses (once) a different worker ID, and checks whether that worker is waiting for work (if not, it gives up for on communication for now). If so, it uses a compare-and-swap operation (which might be racing with other workers trying to assign the same worker a task) to attempt to atomically update the worker's communication cell with the task ID to send, removing this from its own queue if this operation is successful.

Tasks for version 1

(h) Prove that all functions provided for this scheme are memory-safe / crash free (as for task (c) above).

(i) Prove that the same properties (d)–(g) as for version 0 for this new sender-initiated concurrent scheme hold (in particular, assuming $\texttt{P} > 1$).

(j) Write a short textual comment labelled `MODULARITY:` explaining to what extent you are able to reuse parts of the *code* and *verification effort/results* between your two different versions.

Version 2: Receiver-initiated Work-stealing The following alternative implementations of the `acquire` and `communicate` functions (along with definitions/state as shown) implement a *receiver-initiated work-stealing* scheme.

```
const int NO_REQUEST = -1 // special "worker ID" value
int r[P] // request cell per worker; initially all NO_REQUEST
const task NO_RESPONSE = -2 // code for 'no task provided yet'
task t[P] // transfer cell per worker; initially all NO_RESPONSE

// called by workers when running out of work
void acquire(int i)
  while true // block until receiving a proper task
    t[i] = NO_RESPONSE // initialize the cell for receiving a task
    int k = random in {0, .., P-1}\{i} // pick random other worker
    if compare_and_swap(&r[k], NO_REQUEST, i) // make a request
      while (t[i] == NO_RESPONSE) // wait for a response
        communicate(i) // reply negatively to incoming queries
      if (t[i] != NO_TASK) // if we obtained a valid task
        add_task(i, t[i])  // get ready to work on that task
        return
      // otherwise, if obtained a negative reply, then try again
    communicate(i) // provide negative reply to incoming queries

// check for incoming steal requests
void communicate(int i)
  int j = r[i] // check our own request cell
  if j == NO_REQUEST // if no request, then nothing to do
    return
  if (empty(q[i]))
    t[j] = NO_TASK // if no task at hand, provide a negative reply
  else
    t[j] = pop_top(q[i]) // else, reply with a task
  r[i] = NO_REQUEST // reset request cell to allow further requests
```

The scheme here is for workers without work to (via `acquire`) first prepare their transfer cell for receiving a task, then pick a random other worker and register a request for work in their request cell. Then workers wait to receive a response: either `NO_TASK` or a task ID (signifying a task transferred to them). All workers are responsible for checking whether they have received requests and responding, by periodically calling the `communicate` function.

Tasks for version 2

(k) Prove that all functions provided for this scheme are memory-safe / crash free (as for task (c) above).

(l) Prove that the same properties (d)–(g) as for version 1 (for $\texttt{P} > 1$).

(m) Write a short textual comment labelled `MODULARITY:` explaining to what extent you are able to reuse parts of the *code* and *verification effort/results* between each of your different versions.