Threads Considered Harmful

Lucian Radu Teodorescu
Threads Considered Harmful

lucteo.ro/pres/2021-accu/
Go To Statement Considered Harmful

Dijkstra, 1968
My second remark is that our intellectual powers are geared to master static relations and that our powers to visualize processes evolving in time are relatively poorly developed.
For that reason we should do (as wise programmers aware of our limitations) our utmost to shorten the conceptual gap between the static program and the dynamic process, to make the correspondence between the program (spread out in text space) and the process (spread out in time) as trivial as possible.

Edgar Dijkstra
Flow Diagrams, Turing Machines And Languages With Only Two Formation Rules

Corrado Böhm and Giuseppe Jacopini
International Computer Science Institute and Istituto Nazionale per le Applicazioni del Calcolo, Roma, Italy

In the first part of the paper, flow diagrams are introduced to represent mental mappings of a set into itself. Although not every diagram is decomposable into a finite number of given base diagrams, this becomes true at a sentential level due to a suitable extension of the given set and of the basic mappings defined in it. Two normalization methods of flow diagrams are given. The first has three base diagrams; the second, only two.

In the second part of the paper, the second method is applied to the theory of Turing machines. With every Turing machine provided with a two-way half-tape, there is associated a similar machine, doing essentially the same job, but working on a tape obtained from the first one by interchanging alternate blank squares. The new machine belongs to the family, elsewhere introduced, generated by composition and iteration from the two machines \( I \) and \( \bar{I} \). That family is a proper subclass of the whole family of Turing machines.

1. Introduction and Summary

The set of block or flow diagrams is a two-dimensional programming language, which was used at the beginning of automatic computing and which now still enjoys a certain favor. As far as is known, a systematic theory of this language does not exist. At the most, there are some papers by Peter [1], Gorn [2], Hermit [3], Ciurupa [4], Rapas [5], Jeover [6], Anser [7], where flow diagrams are introduced with different purposes and defined in connection with the descriptions of algorithms or programs.

This paper was presented in an informal talk at the IFIP International Colloquium on Algorithmic Languages and Automata Theory, Jerusalem, Israel. Preparation of the manuscript was supported by National Science Foundation Grant GP-2868.

This work was carried out at the Istituto Nazionale per le Applicazioni del Calcolo (INAC) in collaboration with the International Computer Science Institute (ICS), under the Italian Consiglio Nazionale delle Ricerche (CNR) Research Group No. 32 for 1962-64.

In this paper, flow diagrams are introduced by the extensive method; this is done to avoid definitions which certainly would not be of much use. In the first part (written by G. Jacopini), methods of normalization of diagrams are studied, which allow them to be decomposed into base diagrams of three types (first result) or of two types (second result). In the second part of the paper (by C. Böhm), some results of a previous paper are reported [8] and the results of the first part of this paper are then used to prove that every Turing machine is reducible into, or to a determined sense is equivalent to, a program written in a language which admits as formation rules only succession and iteration.

2. Normalization of Flow Diagrams

It is a well-known fact that a flow diagram is suitable for representing programs, computers, Turing machines, etc. Diagrams are usually composed of boxes mutually connected by oriented lines. The boxes are of functional type (see Figure 1) when they represent elementary operations to be carried out on an unspecified object \( x \) of a set \( X \), the carrier of which may be imagined concretely at the act of the digits contained in the memory of a computer, the tape configuration of a Turing machine, etc. There are other boxes of preformative type (see Figure 2) which do not operate on an object but decide on the next operation to be carried out, according to whether or not a certain property of \( x \) \( X \) occurs. Examples of diagrams are: \( \Sigma (a, b, \gamma, \alpha, \beta, \epsilon) \) [Figure 3] and \( \Sigma (a, b, \gamma, \alpha, \beta, \epsilon) \) [see Figure 4]. It is easy to see a difference between them. Inside the diagram \( \Sigma \), some parts which may be considered as a diagram can be isolated in such a way that if \( \Sigma (a, b, \gamma, \alpha, \beta, \epsilon) \) do not operate on an object but decide on the
determine, respectively, the diagrams of Figures 5-7, it is natural to write

\[ \Sigma (a, b, \gamma, \alpha, \beta, \epsilon) = (\alpha, \Delta (a, \Gamma (b, \gamma, \alpha))) \]

Nothing of this kind can be done for boxes which \( \gamma \) the same happens for the entire infinite class of similar diagrams

\[ \Omega = \{ \Omega_1, \Omega_2, \Omega_3, \ldots, \} \]

where formation rules can be easily imagined.

Let us say that while \( \Sigma \) is decomposable according to subdiagrams \( \Omega_1, \Omega_2, \ldots \), the diagrams of the type \( \Omega_2 \) are not decomposable. From the last consideration, which should be obvious to anyone who tries to write with a
structured programming
this talk

threads are like gotos
reasoning with threads is hard
finding a general alternative to threads
The Problem with Threads
Coordination w/o Synchronization
An Example
A General Solution
Composability & Decomposability
Concurrency Patterns
The Problem with Threads
threads

raw threads + synchronization (locks)
problems with threads

- performance
- understandability
- thread safety
- composability
you are likely to get it wrong!

performance
understandability
thread safety
composability
1. cost of locking
synchronization

==

locks

==

bottlenecks
I've often joked that instead of picking up Djikstra's cute acronym we should have called the basic synchronization object "the bottleneck"

David Butenhof
bottleneck  noun

Definition of bottleneck (Entry 2 of 3)

1  a  : a narrow route
   b  : a point of traffic congestion

2  a  : someone or something that retards or halts free movement and progress
   b  : IMPASSE
   c  : a dramatic reduction in the size of a population (as of a species) that results in a decrease in genetic variation

3  : a style of guitar playing in which glissando effects are produced by sliding an object (such as a knife blade or the neck of a bottle) along the strings — called also bottleneck guitar
performance

threads

synchronization

bottlenecks
locks do not scale

chain of locks
prolonged pauses
2. adding a lot of threads
multiple threads on the same core

2 x 1 second
1 core
2 threads, 1 core
2 threads, 1 core
good?
alternative
example 1
Happy families are all alike; every unhappy family is unhappy in its own way.
he
vigorously
embraced
the
pillow
on
the
other
side
and
buried
his
face
in
it
Happy families are all alike; every unhappy family is unhappy in its own way.

he vigorously embraced the pillow on the other side and buried his face in it.
ready?
Happy families are all alike; every unhappy family is unhappy in its own way.

he vigorously embraced the pillow on the other side and buried his face in it.
results

individual texts
~3+3 s

interleaved texts
~10s
example 2

task = 1 step
serial execution of steps = walking
concurrent execution = walking with tied legs
# threads = # cores

~ for CPU intensive tasks ~
3. **composability**

one cannot simply compose two programs

deadlocks, livelocks

performance problems
essential in sw. eng.
threads & locks

low level primitives

like goto
A General Solution
concurrency vs parallelism

goal: design for expressing the concurrency
a single-threaded application
a single-threaded application
a single-threaded application
a single-threaded application
using Task = std::function<void()>::;
```cpp
std::vector<Task> tasks;

int r1, r2, r3;
tasks.emplace_back([&]() { r1 = f1(); });
tasks.emplace_back([&]() { r2 = f2(r1); });
tasks.emplace_back([&]() { r3 = f3(r2); });
tasks.emplace_back([&]() { f4(r3); });

for (const auto& t: tasks)
    t();
```
a multi-threaded application
tasks

independent units of work
constraints instead of locks
a task is **enqueued** when it can safely be executed

... after predecessors/constraints are done

after a task is done it checks if there are successors to enqueue
two key moments

- before the task is started (caller)
- after a task is finished (callee)
theoretical results

- all concurrent algorithms
- safety ensured
- no need for locks
- high efficiency for greedy algorithm
- high speedups
high performance

overhead of tasks management can be small

tasks are independent by design

$$S_p \geq \frac{N}{K + \frac{N-K}{P}}$$
high performance

overhead of tasks management can be small

tasks are independent by design

\[ S_p \geq \frac{N}{K + \frac{N - K}{P}} \]

\[ S_{1000} = 500.25 \]
\[ S_{10} = 9.91 \]

\[ N = 1000 \]
\[ K = 1 \]
Coordination without Synchronization
1. mutexes
two threads with a mutex

- non-contending
- waiting for the lock
- holding the lock
a possible solution
a possible solution (2)
a better notation

color-coded restrictions
a task with restrictions

cannot run in parallel
with a task colored like the restriction
dynamic representation
example

```cpp
concore::serializer my_ser;
backup_engine my_backup_engine; // global resource

void trigger_backup(app_data data) {
    my_ser.execute([=]{ my_backup_engine.save(data); });
}
```
global_executor

worker
worker
worker

max 1 task at a time

queue
implementation details

keep a queue of tasks to be enqueued
keep track if we are executing a task
when finishing executing a task, enqueue the next task
dynamic representation

pass to global
global_executor

pass to
task_serializer
task executors

global_executor — a task is executed as soon as a worker is free

task_serializer — execute at most one task at a given time
2. read–write mutexes
read–write mutex
same notation

- single color restrictions
- dual color restrictions
problem representation
```cpp
concore::rw_serializer my_ser;
backup_engine my_backup_engine; // global resource

void get_latest_backup_info(std::function<void (backup_info)> f) {
  my_ser.reader().execute([&f] {
    // query backup data
    auto data = std::move(my_backup_engine.get_info());
    // call the callback
    f(std::move(data));
  });
}
```
One W task at a time multiple R tasks
task executors

global_executor — a task is executed as soon as a worker is free

task_serializer — execute at most one task at a given time

rw_task_serializer — restrictions between R and W tasks
3. semaphores
semaphore, count 2
improving notation

C_1 \rightarrow C_2

max parallel count
problem representation
```cpp
concore::n_serializer my_ser{10};
concore::concurrent_queue<backup_engine> my_backup_engines; // 10

void trigger_backup(app_data data) {
    my_ser.execute( [=] {
        // acquire a free backup engine
        backup_engine engine;
        bool res = my_backup_engines.try_pop(engine);
        assert(res);
        // do the backup
        engine.save(data); // assume no exceptions
        // release the backup engine to the system
        my_backup_engines.push(std::move(engine));
    });
}
```
The diagram shows a system architecture with a `global_executor` at the center, which distributes tasks to multiple `worker` nodes. The `n_task_serializer` receives tasks and serializes them, placing them in a `queue`. The `global_executor` manages the distribution of tasks, ensuring that no more than `max N` tasks are processed at a time.
task executors

global_executor — a task is executed as soon as a worker is free

task_serializer — execute at most one task at a given time

rw_task_serializer — restrictions between R and W tasks

n_task_serializer — execute at most N tasks at a given time
~ results so far ~
systematic way

raising the abstraction level
NO LOCKS

we have a systematic way of avoiding locks
no blocking needed
maximizing throughput

make sure you have enough tasks in the system
just to be clear

Locks
- can block threads
- reduce throughput

Serializers
- do not block
- throughput is ok if enough tasks
Composability and Decomposability
1. task systems are composable

inner constraints are kept
may require some extra constraints

no different than regular software
2. decomposing tasks

e.g., decomposing serializer tasks

serialize all “user account” actions

some tasks can be decomposed
user account serializer

change username
change password
profile activity
notification settings
user account serializer
user account serializer

- change username
- change password
- profile activity
- notification settings
user account serializer

change username ➔ change password ➔ profile activity ➔ notification settings
user account serializer

- change username
- change password
- profile activity
- notification settings
- query access
- query trans.
- query tickets
task continuations

starting the task → task work → continuation
task continuations

starting the task  →  continuation

task work
new task

using Task = std::pair<
    std::function< void() >, // work
    std::function< void() > // continuation
>;
serializer
serializer
1. exchange continuations
serializer

1. exchange continuations
1. exchange continuations
serializer

2. call new logic
3. remove old logic
3. remove old logic
serializer

done
```cpp
void profile_activity(ProfileActivityResPtr res) {
    // 1. exchange continuations
    auto cont = concore::exchange_cur_continuation();

    // 2. call new logic (spawn tasks)
    concore::task lastTask{[res] { aggregate_results(res); }, {}, cont};
    concore::finish_task ft(std::move(lastTask), 3);
    auto event = doneTask.event();
    concore::spawn([event, res] {
        query_access(res);
        event.notify_done();
    });
    concore::spawn([event, res] {
        query_transactions(res);
        event.notify_done();
    });
    concore::spawn([event, res] {
        query_tickets(res);
        event.notify_done();
    });
}

my_serializer.execute([r] { profile_activity(r); });
```
top-down approach
An Example
h264dec

video decompression software
part of StarBench parallel benchmark suite
basic flow

parse frame

decode slice macro blocks

decode slice entropy

output frame

line decoding
concurrency constraints

- stages need to processed in order for a frame
- parsing needs to be in order
- decode macro-blocks needs to be in order
  - a line depends on the previous line
- frame output needs to be in order
1. Starbench pthreads solution

included in the benchmark
threads

1 parser thread
N threads for entropy / macro-blocks
1 reorder thread (!)
1 output thread
inter-frame dependency

busy wait for the previous macro-block

```c
for(i=0; i< mb_width; i++){
    if (frames || line>0){
        while (rle->mb_cnt >= rle->prev_line->mb_cnt - 1);
    }
    h264_decode_mb_internal( d, d->mrs, s, &m[i]);
    rle->mb_cnt++;
}
```
2. concore solution

concore

Core abstractions for dealing with concurrency in C++

About

concore is a C++ library that aims to raise the abstraction level when designing concurrent programs. It allows the user to build complex concurrent programs without the need of manually controlling threads and without the need of (blocking) synchronization primitives. Instead, it allows the user to "describe" the existing concurrency, pushing the planning and execution at the library level.

We strongly believe that the user should focus on describing the concurrency, not fighting synchronization problems.

The library also aims at building highly efficient applications, by trying to maximize the throughput.
general approach

pipeline for general flow
tasks for macro-block lines
pipeline

- parse
- entropy
- macro-block
- output
pipeline

concore::task_group group = concore::task_group::create();

concore::pipeline<DecFrame> process{h->threads, group};

auto in_order = concore::stage_ordering::in_order;
auto conc = concore::stage_ordering::concurrent;

process.add_stage(in_order,
    [&process](DecFrame& frm) { stage_parse(frm, process); });
process.add_stage(conc, &stage_decode_slice_entropy);
process.add_stage(in_order, &stage_decode_slice_mb);
process.add_stage(in_order, &stage_gen_output);

// Push the first frame through the pipeline
process.push(DecFrame{0, &ctx});

// Wait until we process all the pipeline
concore::wait(group);
macro-block dependencies
processing macro-blocks

struct process_matrix {
    using cell_fun_t = std::function<void(int x, int y)>;

    void start(int w, int h, cell_fun_t cf, concore::task&& donet) {
        width = w;
        height = h;
        cell_fun = cf;
        done_task = std::move(donet);
        ref_counts.resize(h * w);
        for ( int y=0; y<h; y++ ) {
            for ( int x=0; x<w; x++ )
                ref_counts[y*w + x].store( x==0 || y==0 || x==w-1 ? 1 : 2 );
        }

        // Start with the first cell
        concore::spawn(create_cell_task(0, 0));
    }
}
processing macro-blocks

```cpp
... 
concore::task create_cell_task(int x, int y) {
    auto f = [this, x, y] { cell_fun(x, y); };
    auto cont = [this, x, y] (std::exception_ptr) {
        if (y < height - 1 && x > 0) // Spawn bottom task
            unblock_cell(x - 1, y + 1);
        if (x < width - 1) // Spawn right task
            unblock_cell(x + 1, y, false);
        if (y == height - 1 && x == width - 1) // Finish?
            concore::spawn(std::move(done_task), false);
    };
    return concore::task{f, {}, cont};
}

void unblock_cell(int x, int y, bool wake_workers = true) {
    int idx = y * width + x;
    if (ref_counts[idx]-- == 1)
        concore::spawn(create_cell_task(x, y), wake_workers);
}
```
decomposition

MB₁ → O₁ → MB₂ → O₂

MB₁ → O₂ → MB₂ → O₁
decomposition

@LucT3o
void stage_decode_slice_mb(DecFrame& frm) {
...
// This will be broken into multiple tasks. Exchange continuation.
auto cont = concore::exchange_cur_continuation();
auto grp = concore::task_group::current_task_group();
concore::task end_task{[] {}, grp, std::move(cont)};

auto chunk_fun = [&frm] (int x, int y) {
    decode_slice_mb_chunk(frm.global_ctx, *frm.frame_data, x, y);
};
int width = h->mb_width / mb_line_chunk_size;
fd.mb_processing.start(width, h->mb_height, chunk_fun, std::move(end_task));
}
Serial execution
pthreads
concore
Ideal
more performance

not a lot of (naive) concurrency
small tasks → more overhead
limited concurrency (1)
limited concurrency (2)
lines of code

- serial: 200
- concore: 400
- pthreads: 600
results

easy to write
high-level concurrency abstractions
efficient

https://github.com/lucteo/h264dec-concore
Concurrency Patterns
1. create concurrent work
```cpp
void start() {
    initComponents();
    concore::spawn([]{
        loadAssets();
    });
    concore::spawn([]{
        initializeComputationEngine();
    });
}
```
2. delayed continuation
```cpp
void handleResponse(HttpResponseData respData, HandlerType handler) {
    // the work for this task: process the response
    HttpResponse resp = respData.toResponse();
    // create a continuation to handle the response
    concore::task cont{[resp = std::move(resp), handler] {
        handler(resp);
    }};
    concore::spawn(std::move(cont));
}

void httpAsyncCall(const char* url, HandlerType handler) {
    // does HTTP logic, and eventually async calls handleResponse()
}

void useHttpCode() {
    // the work to be executed as a continuation
    HandlerType handler = []([HttpResponse resp) {
        printResponse(resp);
    };
    // call the Http code asynchronously, passing the continuation work
    httpAsyncCall("www.google.com", handler);
    // whenever the response comes back, the above handler is called
}
```
3. join
```cpp
concore::finish_task doneTask([]{
    listenForRequests();
}, 2); // waits on 2 tasks

// Spawn 2 tasks
auto event = doneTask.event();
concore::spawn([event] {
    loadAssets();
    event.notify_done();
});
concore::spawn([event] {
    initializeComputationEngine();
    event.notify_done();
});
// When they complete, the done task is triggered
```
4. fork-join
template <typename F>
void conc_apply(int start, int end, int granularity, F f) {
    if (end - start <= granularity)
        for (int i = start; i < end; i++)
            f(i);
    else {
        int mid = start + (end - start) / 2;
        auto grp = concore::task_group::create();
        concore::spawn([=] { conc_apply(start, mid, granularity, f); }, grp);
        concore::spawn([=] { conc_apply(mid, end, granularity, f); }, grp);
        concore::wait(grp);
    }
}
5. task graphs
std::shared_ptr<RequestData> data = CreateRequestData();

// create the tasks
concore::chained_task t1{[data] { ReadRequest(data); }};
concore::chained_task t2{[data] { Parse(data); }};
concore::chained_task t3{[data] { Authenticate(data); }};
concore::chained_task t4{[data] { StoreBeginAction(data); }};
concore::chained_task t5{[data] { AllocResources(data); }};
concore::chained_task t6{[data] { ComputeResult(data); }};
concore::chained_task t7{[data] { StoreEndAction(data); }};
concore::chained_task t8{[data] { UpdateStats(data); }};
concore::chained_task t9{[data] { SendResponse(data); }};

// set up dependencies
concore::add_dependencies(t1, {t2, t3});
concore::add_dependencies(t2, {t4, t5});
concore::add_dependency(t4, t7);
concore::add_dependencies({t3, t5}, t6);
concore::add_dependencies(t6, {t7, t8, t9});

// start the graph
concore::spawn(t1);
6. serializers
7. concurrent for
std::vector<int> ids = getAssetIds();
int n = ids.size();
std::vector<Asset> assets(n);

concore::conc_for(0, n, [&](int i) { assets[i] = prepareAsset(ids[i]); });
template< class ExecutionPolicy, class ForwardIt, class UnaryFunction2 >
void for_each( ExecutionPolicy&& policy, ForwardIt first, ForwardIt last, UnaryFunction2 f );
8. concurrent reduce
```cpp
std::vector<Resource> res = getResources();

auto oper = [&](int prevMem, const Resource& res) -> int {
    return prevMem + getMemoryConsumption(res);
};
auto reduce = [](int lhs, int rhs) -> int {
    return lhs + rhs;
};
int totalMem = concore::conc_reduce(res.begin(), res.end(), 0, oper, reduce);
```
template<
class ExecutionPolicy,
class ForwardIt, class T, class BinaryOp, class UnaryOp>
T transform_reduce(ExecutionPolicy&& policy,
    ForwardIt first, ForwardIt last,
    T init, BinaryOp binary_op, UnaryOp unary_op);
9. concurrent scan
std::vector<FeatureVector> in = getInputData();
std::vector<FeatureVector> out(in.size());

auto op = [](FeatureVector lhs, FeatureVector rhs) -> FeatureVector {
    return combineFeatures(lhs, rhs);
};

concore::conc_scan(in.begin(), in.end(), out.begin(), FeatureVector(), op);
template< class ExecutionPolicy, class ForwardIt1, class ForwardIt2, class BinaryOperation, class T >
ForwardIt2 inclusive_scan( ExecutionPolicy&& policy, ForwardIt1 first, ForwardIt1 last, ForwardIt2 d_first, BinaryOperation binary_op, T init );
high-level concurrency abstractions

no more low-level primitives
Conclusions
My second remark is that our intellectual powers are geared to master static relations and that our powers to visualize processes evolving in time are relatively poorly developed.
For that reason we should do (as wise programmers aware of our limitations) our utmost to shorten the conceptual gap between the static program and the dynamic process, to make the correspondence between the program (according to the concurrent design) and the process (execution over multiple threads) as trivial as possible.

Edgar Dijkstra (paraphrased)
Threads Considered Harmful

... but we have an alternative
... a global method
...with no locks
threading primitives

pushed down to the framework level
systematic way

raising the abstraction level
composable/decomposable
no excuse for raw threads and locks
http://nolocks.org
Thank You

@LucT3o
lucteo.ro
nolocks.org