Course report on parallel preflow-push

Labs in EDAN26 Multicore programming

student1 and student2
Lund University

September 24, 2021

Contents

1 Performance results

Lab	Language*	Synchronization	Time / s	Note
1	Akka	actors		
2	Java	locks		
2	C	locks		
3	C	barriers		
4	C	atomic		
5	Rust	locks		
6	C	TM		
6	Clojure	TM		

* C is required in Lab 2, and recommended in labs 3, 4 and 6. See Section 5.

2 Competition rules

This is a tentative list of rules for the competition.

1. The deadline for the competition is Sunday October 31st, at 18:00:00.
2. Only programs based on the preflow push algorithm are permitted.
3. If you would violate any constraints in the algorithm, you must prove that it is correct to do so and not just happens to work on the test cases. An example of a violated constraint would be to allow the height of a node to have any value (which would not be permitted since the algorithm will not work then).
4. It is obviously permitted to be inspired by published articles.
5. It must be written in ISO C plus Pthreads, meaning no GNU extensions and no machine code, except that it is permitted to use hardware transactional memory as in Lab 6 (despite not being ISO C).
6. No data-races are permitted.

3 More theory about preflow-push

The preflow-push algorithm is almost always presented with a while loop that iterates until no node (except the sink) has a positive excess preflow. We used the same in EDAF05.

Nils Ceberg made the following nice observation, that instead of finding out when only the sink has a positive preflow, it is sufficient to check if the flow out from the source is equal to the flow in to the sink. This check can be done *after* the source has done its initial pushes to its neighbors. This works since after the initial pushes the flow out from the source decreases monotonically, and the flow in to the sink always increases monotonically (since it keeps all flow it gets and never pushes it away). Another way to see this, is that the flow out from the source must be equal to the sum S of all excess preflows for all nodes except the source, so when the flow out from the source is equal to the flow in to the sink, only the sink contributes to S so the excess preflow of all other nodes must be zero, at which point the algorithm should terminate, just as with the while-loop condition.

Using this still gives an interesting learning experience in Lab 1, but it will be much simpler than using a controller to check for the termination by counting active nodes. There can be some non-obvious rare races between messages if the controller *only* uses a counter.

4 How to do the labs

- To make lab presentation more efficient, you must have a Discord username that shows your real name (but to ask questions, you can use any username).
- Except for Lab 6, you can do the labs on Mac, Linux, or Windows with the Linux subsystem, but they should be presented power.cs.lth.se, below called Power.
- Some functionality will only work on Power.
- First download the Tresorit directory you got an email about, or copy the directory /opt/tresorit/preflow on Power to your account there (or with scp to any computer).
- Check out the sequential C program preflow.c in lab0 and the pseudo code in preflow.pdf in Tresorit and make sure you understand it. If not, ask at Discord or book a Zoom meeting with Jonas. Zoom is booked at `calendly.com/forsete`
- Then read the language and general requirements below and the specific requirements and questions for a lab. Each lab has hints at the end of this pdf which you might want to check out.
- No report should be written except that you should fill in the performance numbers at the first page (with suitable notes for yourself only to keep track

of your best solution so far). Edit labs.tex from Tresorit in the labs/doc directory (which has two support files).

5 Language requirements

It is required to have parallelized preflow push in Scala/Akka (lab 1), Java, C (lab 2), Rust (lab 5) and Clojure (lab 6). In labs 3, 4, and 6 you can use any language with support for automatic data-race detection (such as C and C++), and:

- Lab 3: barriers (or implement your own barrier)
- Lab 4: atomic variables
- Lab 6: hardware transactional memory

C is a safe choice :)

6 Debugging and help from teachers

As a general rule, it is *not* expected that teachers will debug your code, partly because it can be too time-consuming and partly because it is an essential skill to develop. If there is sufficient time on a lab session, of course you can ask the teachers for debugging help. It is likely you will find C with the Google sanitizer plus printing lots of output the most efficient, plus using gdb if your program crashes (see the course book). Instead we are happy to look for ”common mistakes” and give general hints on debugging parallel preflow push in Scala/Akka, Java, C, Rust and Clojure. If you need help to move forward, you can always book a Zoom meeting with Jonas (see above).

It can also be a good idea to write a question in a text-channel (lab or language specific) plus answering other questions! Feel free to suggest additional language-specific channels.

7 Lab requirements and questions

The teachers will ask you questions including these but it is not required to write the answers in the pdf.

General requirements

1. Which lab solutions should be uploaded to Forsete (and which URL) will be decided later but at least the C labs should be uploaded.

2. You should present the labs in Discord (with a username that shows your real name to make presentation more efficient) and be logged in to the Power machine (see email or tresorit).

3. When you type the command `make` on Power, you should get output `PASS` and the execution time. Sometimes you need to type e.g. `make par` or modify the `makefile`.

4. Fill in the table above with the *fastest* execution time of a few runs (fastest since then your program was not so much disturbed by other things — but in other situations it can be more relevant to measure average or slowest).

5. All source code should be written by yourself or your lab partner, in which case you should understand it as if you had written it yourself.

6. No data-races or potential deadlocks are permitted, as reported by Google sanitizer or Helgrind (use smaller inputs for Helgrind).

7. You are allowed to use any library functions available in the language you use except if there is one for maxflow.

8. In the C/Pthreads labs, it is only in the lab on transactional memory that you may use non-standard extensions of ISO C, as shown in the hint.

9. You are not allowed to use timeouts (or ”periodic checks”) to decide when your program ”probably” has completed.

10. For the competition: you can use any algorithm for maxflow, as long as you have written it yourself. Feel free to get ideas from the research literature. It can be useful to select alternatives in the program based on values from the input, but that is not permitted, except that you may select what to do based on the number of nodes and edges. It is also permitted to determine general characteristics about the input such the length of a shortest path from source to sink (if that would be relevant), or similar, but it is not permitted to check e.g. if there is an edge between nodes v_{124} and v_{129} and then select an alternative optimized for that input.

Lab 1: Scala/Akka

Purposes

You will learn about:

- What actors are and how they communicate
- Guarantees about the order in which messages arrive
- How to know when an Akka program has completed (one example)

Relevant sections in the book

See lecture 2, since the book does not cover Scala/Akka but if you are new to terminals, Appendix A will be useful (you can find it in Tresorit as well).

Requirements

1. Each node must be one actor.
2. You are only allowed to share an edge object between two nodes but no other information (such as storing their heights somewhere easily accessible from any node or from a neighbor).
3. The only way to get information from another actor is to ask for it using a message and no shared data is allowed (except edges).
4. It is not permitted to use ? between nodes but if needed with a controller for ”critical messages”.

   The reason is that the default way of thinking with actors should be ”send a message and wait for any message”, i.e. with !

   (Motivation: without forbidding ? between nodes, it is ”too easy” to use messages as normal method/function calls (but that limits concurrency normally). If you think of a chat server when it sends a text to many phones, it would be too slow to send one at a time with ?)
5. If you use a controller, it is not permitted to let it periodically wake up nodes. Instead it should either keep track of which node is active, or solve it in some other way.

Note

1. There is no performance requirement for Scala/Akka.

Questions

1. If you use a controller, how can it know that no node is active? If you don't use a controller for that, how do the actors agree on terminating the algorithm?
2. How many push messages can a node have in progress concurrently in your solution? (one is OK)
3. How do you ensure that a node never gets a negative excess preflow? (except possibly for the source depending on how you implement it)
4. With millions of nodes in a graph, it would be bad to have one JVM thread per actor, so how is it done?
5. Explain your source code.
6. Did you find a way to improve the performance of your program over the first correct version?
7. For which types of applications would you consider using Akka and why?

Getting started on Power

If you have a working `preflow.scala`, then the following commands can be used to run it on the Power.

```
scp preflow.scala stilid@power.cs.lth.se:
ssh stilid@power.cs.lth.se
cp -r /opt/tresorit/preflow/labs .
mv preflow.scala labs/lab1
cd labs/lab1
make
```

See also appendix-A.pdf and `ssh-copy-id` for how to avoid typing your password too many times. Specifically, it creates a file on your local computer and copies it e.g. to Power so when you want to login to Power, no password is needed. The appendix is from the course book and can be found in Tresorit.

Lab 2: Java and C with locks

Purposes

You will learn about:

- How to avoid deadlocks in multithreaded programs
- How to detect data-races in C automatically
- How to detect risks for deadlocks in C automatically
- The importance of a load-balancing
- The importance of a suitable task granularity that matches the structure of the program in order to reduce most of the overhead from thread synchronization. Recall that a task is the job performed as a ”work-item” by a thread such as what is done with one node with excess preflow.

Relevant sections in the book

Chapter 18 and sections 6.5 and 6.9.1 if your C program crashes.

Requirements

1. Parallelize preflow-push both in Java and C with locks and possibly with condition variables (see hints about which language may be preferable to do first).
2. The concurrency must scale with the number of nodes and threads so you cannot use for instance only one lock for the whole program. Use for example one lock per node.
3. Check your C program both with the Google Thread Sanitizer (often called tsan) and Helgrind.
4. Print out a message when each thread terminates.
5. What fraction of the execution time is wasted on synchronization?

   To measure this, you can use the `operf` and `opreport` commands. See the course book or `operf.pdf` in Tresorit. On Power you can do:

   ```
   gcc -g -O3 -pthread preflow.c
   operf ./a.out
   opreport -t 1 -l a.out
   ```

6. Collect and present statistics about load-balancing, such as for instance how many nodes each thread has processed.

Optional

On the Power computer, you can make very accurate timing-measurements using the so called Timebase Register, which is incremented regularly (not every clock cycle to avoid overflow but very often). It is not compulsory for the labs to use it, but if you want to, do `#include "timebase.h"` and in `main` use:

```
double          begin;
double          end;

init_timebase();
```

And measure time as:

```
begin = timebase_sec();
f = preflow();
end = timebase_sec();

printf("t = %lf s\n", end-begin);
```

You may want to modify the `makefile` to:

```
gcc -g -O3 -pthread preflow.c tbr.s timebase.c
```

Or copy `/opt/tresorit/preflow/labs/lab2/c/makefile` on Power.

Questions

1. How do you avoid data-races and deadlocks?
2. What is your approach to load-balancing and how does it succeed?
3. Is your parallel program faster than the sequential from Lab 0, and what do you think is the reason for this?
4. Did you notice any big performance difference between Java and C? (in principle, either can be fastest and the JVM and the optimizing compiler are important factors, and how you use the languages).
5. Explain your source codes.
6. Did you find a way to improve the performance of your programs over the first correct version?

Lab 3: Barriers

Purposes

You will learn about:

- How running threads in phases sometimes can reduce synchronization overhead and increase performance.
- How to use barriers.

Relevant sections in the book

The same as for Lab 2.

Requirements

1. Either (1) modify your C program from Lab 2, or (2) use any other language of your choice that supports data-race detection, so that the threads run in two phases:
    - a first phase to let each thread decide on how much its nodes should push to a neighbor (or that a relabel is needed), and
    - a second phase with only one thread doing work to update the excess preflows and heights
2. Check your program for data-races as in Lab 2.
3. What fraction of the execution time is wasted on synchronization?
4. Collect and present statistics about load-balancing.
5. Your program should now be faster than the sequential from Lab 0, which gets a score of 35 at Forsete.

Questions

1. Does the use of phases affect the possibility to do good load-balancing, and why, and if it does, to what extent is it significant?
2. Explain your source code.
3. Can you distribute the nodes in a fair way for load-balance?
4. Did you find a way to improve the performance of your program over the first correct version?

Lab 4: Atomic variables

Purposes

You will learn about

- The overhead of using atomic variables over non-atomic variables.
- When this overhead may save time from other parts of a program.

Relevant sections in the book

Chapter 5, sections 7.18, and 13.16.

Requirements

1. Make a copy of the sequential C program from Lab 0 and change the type from `int` to `atomic_int` for each variable in the `node_t` and `edge_t` types., and start the file with `#include <stdatomic.h>`
2. Measure how this affects the execution time (without adding any threads).
3. Find an explanation for why or why not this affects the execution time by looking at the assembler code. See hint for this lab.
4. Modify your program from Lab 3 (or Lab 2 if the language used in Lab 3 has no support for atomic variables) so that it uses both barriers and atomic variables. It is not sufficient to just change all `int`s to `atomic_int`s. You need to somehow exploit the atomic variables. One option is to accumulate incoming pushes in a new, atomic, variable in phase 1, and then add that to the excess preflow in phase 2, but there are more alternatives.
5. Your program should still be faster than the sequential from Lab 0.
6. Test your program both with sequential consistency and relaxed memory order. Depending on how how complex context you use atomic variables in, it may be impossible to use only relaxed memory order but for example accumulating incoming pushes (incrementing a counter where only the sum at the end matters and not in which order the counter was incremented) it is safe for relaxed memory order. In other contexts you may need to also use memory order release and acquire.

Questions

1. What is your explanation to requirement 3?
2. With only one thread doing work in phase 2 in Lab 3, can you move some of that work to phase 1 instead?

3. Which memory order is used by default for compound assignment operators (such as +=) ? (in case you don't use C/C++, your answer may differ from these languages)

4. Will you get a data-race if you switch to `memory_order_relaxed` in C/C++? Why or why not?

5. Can you notice any performance difference when using (1) sequential consistency memory order and (2) relaxed memory order? Why or why not do you think?

6. Explain your source code.

7. Did you find a way to improve the performance of your program over the first correct version?

8. What is the purpose of each of memory order release and memory order acquire?

Lab 5: Rust

Purposes

You will learn how Rust is used when

- different pointers need to point at an object: move and borrow
- multiple pointers/threads need to use an object: atomic reference counters
- moving objects when creating threads

Relevant sections in the book

See lecture 7 since Rust is not covered in the book.

Requirements

1. It is not permitted to use the keyword `unsafe` which essentially makes the program as unsafe as C and C++. The whole point with Rust is that the compiler should give guarantees about the safety of the program but with `unsafe` it cannot.
2. Make a sequential version of preflow-push
3. Make a parallel version of preflow-push

Installing the Rust compiler macOS

If you have not yet done so, visit `https://brew.sh` and follow the instructions. With `brew` you can install numerous useful command-line tools on your Mac.

```
brew install rust
```

Installing the Rust compiler on Ubuntu and Windows

If you get the error message: `/bin/sh: cargo: command not found` when you run `make`, you need to first install the Rust compiler:

```
sudo apt install rustc
```

Note

1. *Tentatively*: there is no performance requirement for this lab.
2. It is OK to use one shared set of active nodes. Optionally let each thread have its own set of active nodes.

Optional

1. Make the parallel version faster than your program from Lab 4 but it is not at all certain you will succeed — rather, it is quite unlikely :)

Questions

1. Explain Rust's rules about moving and borrowing objects.
2. Explain atomic reference counters.
3. Explain your source code for your two programs.
4. What do you think about parallelizing Rust code as compared with Java and C?

Lab 6: TM, OpenMP, and parallelizing compilers

Purposes

You will learn about:

- hardware transactional memory in C
- software transactional memory in Clojure
- OpenMP
- parallelizing compilers

Relevant sections in the book

Chapter 20, and you may want to check out chapter 16 on optimizing compilers if you are curious but it is not part of the course really, and lecture 8 about Clojure.

Requirements

1. Modify one of your C programs to use hardware transactional memory.
2. Make a sequential Clojure program for preflow-push.
3. Parallelize your Clojure program.
4. Parallelize matrix multiplication using OpenMP (copy `mm.c` to a new file and add `-fopenmp`) and measure the speedup of the sequential version. Make sure you have read the hint about accesses to the `c`-matrix.
5. Compare the performance of `mm.c` using the commands below. Time a.out after each compilation. You may want to press CTRL-C after 10 s if a.out has not yet finished.

```
clang -mcpu=power8 -O3 mm.c
gcc -fexpensive-optimizations -mcpu=power8 -O3 mm.c
gcc -ftree-parallelize-loops=80 -fexpensive-optimizations -mcpu=power8 -O3 mm.c
pgcc -tp=pwr8 -O4 mm.c -Mconcur=allcores
xlc -qarch=pwr8 -O5 -qsmp -qhot=level=2 mm.c
```

Note

1. There is no performance requirement for Clojure.
2. It is OK to use one shared set of active nodes. Optionally let each thread have its own set of active nodes.

Questions

1. Why is it more reasonable to use software transactional memory with Clojure than with C, C++ or Java?
2. How does Power detect conflicts between hardware transactions?
3. Why can you not use I/O in a transaction, and which instructions can be used on Power if you need to print something?
4. Explain your source codes.

8 Scala/Akka background

Lab input files

We assume you have copied the `multicore` directory from Tresorit. All labs use the same input files located in `multicore/data`. Each lab has its own check-solution script which will use these common input files. The first tests are from Lab 6 of EDAF05, except that we will not remove any edges (so c and p are ignored from that lab). There are also other tests. The graphs have n nodes, numbered as $0..n-1$ with the source being 0 and the sink $n-1$.

`sbt`

The Scala build tool, `sbt`, can compile and run Scala programs. It downloads required libraries as needed. Since it compiles all Scala files in the current directory, it is usually a good idea to have only one Scala file in `multicore/lab1`. Now give the commands:

```
cd multicore/lab1
make
```

If all is properly installed (such as Java), this will compile and run a file `preflow.scala` in the lab 1 directory.

The program `make` looks for a file called `makefile` which contains commands to execute, and in this case, it simply runs `sbt` with an input file `i` as standard input using so called I/O redirection, i.e., like this

```
./sbt run < i
```

The file `i` contains a small example graph:

```
3 2 0 0
0 1 10
1 2 2
```

The input format is as follows: the first line contains four values: n, m, c, and p where n is the number of nodes, m is the number of edges, and the ignored c and p. Then follow m lines, each with an edge. An edge is specified as u, v, and a positive flow capacity. The edges are undirected and there can only be one edge between two specific nodes u and v in the input.

In this example, the source will first push 10 to node 1 which can push 2 to the sink, and then, eventually, push back 8 to the source, so the maximum flow is 2 and the program should print `f = 2`. However, since the Scala program is incomplete, it will instead print `f = 0`.

Akka

Actors in different languages are usually very similar. We will now describe them in general terms for Akka. An actor is like a thread in that it can do things on its own, and an actor does one of two things:

- waits for a message to arrive, and
- processes the next message that it received.

Some important facts include:

- A message never interrupts and actor. The actor processes the current message and after that waits for new messages.
- To send a message to another actor, the sender usually does not wait for a reply (but it is possible, as we will see).
- If an actor sends multiple messages to another actor, all these messages will arrive in the same order.
- There is no other order between messages.
- It is possible to avoid data-races by regarding all received data as read-only.

For the lab, each node should be an actor and this is achieved by extending the Actor class:

```
class Node(val index: Int) extends Actor {
        var     e = 0;                          /* excess preflow.
        var     h = 0;                          /* height.

        /* ... */
}
```

It is impossible to access an actor's private variables. Ask for their current value using a message.

In the lab, there is one special actor which manages the computation, and then one actor for each node.

To send a message, a reference to actor is needed plus the message. For example:

```
node(t) ! Excess        /* ask sink for its excess preflow */
```

Here `node(t)` is a reference to the sink node, `!` is the operator to send a message without waiting for a reply, and `Excess` is a message. This message is the name of a type declared as

```
case object Excess
```

A message without a parameter should be written as object as here. The `case` is similar to a case-statement in Java or C, i.e. code to execute is selected based on some value (as in a switch-statement).

To receive this message, a node should have a function called `receive` defined e.g. as

```
def receive = {

case Excess => { sender ! Flow(e) /* send our current e. */ }

/* other cases */

}
```

The syntax means: match a case, and then do what is to the right of the `=>`. To ”match”, the type of the message and the type of the case should be similar. A case with a lower case identifier would introduce a variable that matches anything so be careful not to write:

```
case excess => { sender ! Flow(e) /* reply with our current e. */
```

It matches any message type, so no case below can be matched.

To reply to the same actor which sent us the message, the identifier `sender` is available. With the node’s excess preflow, stored in the attribute `e`, the following type can be used:

```
case class Flow(f: Int)
```

With a parameter, we need to use `class` instead of `object`.

Suppose we want to save the `sender` for future use. We then need a variable of type `ActorRef` and in Scala we can declare it as:

```
var     ret:ActorRef = null     /* ret is just a variable name. */
```

and when we have received a message, we can save its sender as

```
ret = sender
```

The `ActorRef` is like a C pointer or Java object reference. It is not the actor itself. The actor itself may be located in a computer on the other side of the world. For simplicity, we will say ”the actor” when we should have said ”the actor that an `ActorRef` refers to”. We cannot know what type it actually refers to. The only thing we can do is to send a message to it.

Before going into more details, we should look at the (messy) syntax to actually create an actor.

Creating an actor

Before we can create any actor, we need a reference to an actor ”system” like this:

```
        val system = ActorSystem("Main")
```

From the `system` we can create new actors. If the constructor of the actor class takes no parameter, we can use:

```
val control = system.actorOf(Props[Preflow], name = "control")
```

This is essentially the same as:

```
Preflow control = new Preflow;
```

in Java. There is a hierarchy of actors so therefore new actors are created not just with `new` but instead from a ”factory” and this results in the syntax above. In addition, each new actor needs to have a name.

For actors with a parameter to its constructor we need to use a different syntax. For example, to give the parameter `index` to a new node (see `Node` above) we can first declare the array of nodes. Note that type of the array element should be `ActorRef` and not `Node` as we might have expected:

```
var     node: Array[ActorRef] = null
```

When we know the number of nodes, n, we can create the array:

```
node = new Array[ActorRef](n)
```

and then the actors as array elements:

```
for (i <- 0 to n-1)
        node(i) = system.actorOf(Props(new Node(i)), name = "v" + i)
```

Note that with a parameter, we *should* use `new`.

Lists in Scala

For more details about lists in Scala, please see the pdf lecture about Scala. A short summary is given here.

An empty list of any type is written as `List()` and to make to source code less cluttered we use:

```
val     nil: List[Edge] = List()
```

To make the source code less cluttered, we use `nil` for an empty edge list.

A list variable should have a type such as `edge` below

```
var     edge: List[Edge] = nil          /* adjacency list. */
```

When we create an edge between u and v, we insert the edge object (which is shared between u and v) first in the node’s adjacency list:

```
case edge:Edge => { this.edge = edge :: this.edge }
```

So the actor is given an edge in a message and puts it in front of the current adjacency list, and then updates the adjacency list to point to this new list link.

Suppose we want to iterate through an adjacency list. One way to do so is:

```
for (r <- edge)
        println("r is an edge: " + r)
```

and here `r` will point to each `Edge` object in turn. If you instead want to iterate through the list in a more manual fashion, you can do as follows:

```
var     p: List[Edge]   = nil
var     e:Edge          = null

p = edge
while (p != nil) {
        e = p(0)
        p = p.tail
}
```

Here the expression `p(k)` means the edge stored k list links away from `p`.

`this` vs `self`

Suppose we have an object of type `Node`, i.e. a type which extends `ActorRef`. To refer to the node, we use `this` as in Java, but to refer to the actor, we instead use `self`. This happens when the controller should send a reference to itself to each node.

To match a message containing an actor, we should use `ActorRef`:

```
case Control(control:ActorRef)  => this.control = control
```

So this case should not have any mention of the class of the object that we want to talk to, i.e., `Preflow`. The reason is that an actor should not know which type it is. It should only talk to other actors. To distinguish between different message types, each containing an actor, we need to use a different case class for each such message type.

Waiting for a reply

To wait for a reply, we use `?` instead of `!` when we send a message:

```
val flow = control ? Maxflow
```

We then create a timeout:

```
implicit val t = Timeout(4 seconds);
```

and wait:

```
val f = Await.result(flow, t.duration)
```

Terminating the program

To terminate the program, we can use:

```
system.stop(control);
system.terminate()
```

9 Hints

Lab 1 hints

In the lab you need to extend the file `preflow.scala` so that it can find the maxflow. How can that be done? A start is:

- Push from the source node to each of its neighbors by sending a suitable message (which does not exceed the remaing flow capacity of the edge, see Lab 0).
- As long as a node, other than the sink, has an excess preflow greater than zero, it should push if it can, and if not, it should relabel.
- Somebody has to figure out when the algorithm has terminated. The easiest, and sufficient for the lab, is to have a controller which keeps track of nodes with excess preflow. If none (other than the source or sink) has excess preflow, the maxflow is the excess preflow of the sink.

What can go wrong with this?

- If u tries to send preflow to v, it should fail if u is not higher than v. Note that after the moment u sends a message to v, the height of v may have increased.
- Active nodes are nodes with excess preflow, and if the controller keeps track of how many are active, it is important to make sure there is never a risk that messages arrive to the controller in an order which would make the controller think none is active when there in fact is a node which soon is going to get some preflow from a neighbor.

Lab 2 hints

- Even if you have never programmed in C before, it may be easier to get a correct program in C than in Java thanks to the Google thread analyzer (tsan) and Helgrind.

  One year a group of two very good students made a quite complex Java solution which had a bug. Instead of helping them, I suggested they should

re-implement the bug for the next week's lab on C. They did and could solve the problem almost immediately using Helgrind (and immediately fix the Java program as well).

- Always start with the simplest possible solution.
- Identify which data two threads may access concurrently such that at least one thread may modify it.

  Try to identify when deadlocks can occur. For instance, if there is an edge (u, v), and one thread t_0 is pushing from u and another t_1 from v, then if t_0 locks u before locking v, and t_1 locks v before locking u, there is a deadlock.

  This situation is detected by tsan and Helgrind. In fact they detect that the threads try to take the locks in the opposite order and therefore warn that there is a risk for deadlock (even if one never happens during a particular execution).

  Two techniques to avoid deadlocks are

  - Impose a lock-order which for instance says that for an edge (u, v), if a thread wants to lock both of u and v, it must lock them in a predefined order (for instance, u first if $u < v$).
  - Use try-lock instead of lock. If the lock was already taken, the thread is not blocked. If a thread is pushing from u and finds v locked, it can skip v and check the next edge instead — but see the next hint about doing a relabel too early.
- It is a bug to relabel a node if it actually could do a push.
- If you for instance (1) have a shared set of nodes with excess preflow, (2) that set is locked when a thread t wants to remove from it (or add a node to it), and (3) t has nothing else to do while waiting, then a condition-variable can be used to wake up t when t should make a new attempt to do something with that set. While this may sound reasonable it will probably lead to bad performance due to contention to the shared set. Can you avoid using synchronization when a thread should take its next node to process?

Lab 3 hints

- Instead of locking nodes and edges as needed, preflow-push can be be divided into two phases such that the heights and edge flows are not modified immediately. We can then have multiple iterations where each iteration consists of two phases.
- In the first phase all threads are allowed to read any node's height, and no relabel is permitted.

- If a relabel is needed, it can be performed in the second phase.
- How many times can the flow of an edge be changed per iteration?
- When should the excess preflow of a node be modified and when should the node be put in some thread's set of active nodes? Recall from Lab 1 on Akka, we say a node is active if it has excess preflow > 0.
- We have edge flow modifications, and for a node modifications of heights, excess preflows, and the next pointer (to the next active node in some thread's set). How can these "commands" be represented and who should perform them? One option is to let each thread use an array of structs so that each struct represents the modifications of one node.
- If you want have an array of structs but you don't know how many elements will be required, you can do essentially as follows:

```
typedef struct {
        int     x;
        int     y;
} something_t;

something_t*            a;      // pointer to the first element
int                     c;      // storage capacity (number of elements)
int                     i;      // next available element

c = 2;
a = malloc(c * sizeof a[0]);    // memory for two elements
if (a == NULL)
        error("no memory");     // error is not a standard function
i = 0;

if (i == c) {
        something_t*    b;

        c *= 2; // double the capacity
        b = realloc(a, c * sizeof(a[0]));
        if (b == NULL)
                error("no memory");
        a = b;
}

a[i].x = x;
a[i].y = y;
i += 1;
```

When the array is no longer needed, you call `free(a)`, but you should not free the old array after using `realloc`, since that is taken care of by `realloc` if needed (it can happen that `realloc` found available memory after the end of `a` so no free was needed).

- Pthread barriers are useful to make sure all threads agree on in which phase work should be done.
- When a barrier is initialized, one parameter is the number of threads T that will use it. Each thread calls the `pthread_barrier_wait` function with the same barrier as an argument. At a call, the first $T-1$ threads are blocked, and when the last thread has made the call, all threads are allowed to resume execution.
- Note that a memory barrier (as in the Linux kernel or Power instruction) is different from a Pthreads barrier.

Lab 4 hints

- To quickly find the machine instructions of some code, you can add a global volatile variable and do something with it that is easy to search for in the assembler file. On Power, you can compile with `-S` and then look in the file with suffix `s`.

```
volatile int hello;
void relabel(graph_t* g, node_t* u)
{
        hello &= 0x1234;
        u->h += 1;
        hello &= 0x5678;

        enter_excess(g, u);
}
```

  Then search for `0x1234` and `0x5678`, and you should see the assembler code for `u->h += 1` in between (the Power instruction for `&=` is `andi.` which means and-immediate and (the meaning of the dot suffix) set condition-codes).

Lab 5 hints

- If you get an error message from `cargo` about an operation not permitted, it may help to remove the `target` directory (which will be re-created automatically when compiling the next time). So if you see:

```
Operation not permitted (os error 1)
```

then do

```
rm -rf target
```

but beware that `rm -rf` is a dangerous command that recursively removes the argument files without asking so be careful.

- Start with studying the `swish.rs` program which simulates a number of swish transactions. You can compile it with

  ```
  rustc swish.rs
  ```

  and run it with

  ```
  ./a.out
  ```

- Then make the sequential preflow-push program complete. It is in the directory:

  ```
  lab5/seq-rust/src/main.rs
  ```

  Type:

  ```
  cd lab5/seq-rust/src
  cargo run
  ```

  which should compile and run an incomplete program.

- Suppose you have a lock for the source node, and want to set its height to *n*, and do

  ```
  let mut source = node_array[s].lock().unwrap();
  source.h = n as i32;

  // continue with initial pushes
  // and then run the while-loop
  ```

  A problem with this code is that the lock taken for `source` is not unlocked until `source` goes out of scope, so if you continue with the rest of the algorithm and still have it locked, you cannot lock it again. Depending on your source code, you may need to unlock by putting the above code in a block:

  ```
  {
          let mut source = node_array[s].lock().unwrap();
          source.h = n as i32;
  }
  ```

  The lock is unlocked when `source` goes out of scope at the `}`.

- Then copy the `seq-rust` directory

  ```
  cp -r seq-rust par-rust
  cd par-rust
  cargo run
  ```

  and parallelize it.

Lab 6 hints

- It is assumed you use a barrier and two phases. Most of the work is done in phase 1 but some of the work done in phase 2 can be moved to phase 1, as with atomic variables. The question is if we can avoid overhead due to locks or atomic variables, or improve load-balance, by using transactional memory.
- Note that with transactional memory we can easily use any normal syntax and not only be limited to what can be expressed with atomic variables. Updating of pointers, for instance, is trivial with transactional memory.
- You should start threads as usual and the difference is how the transactions can be used instead of other synchronization primitives.
- You need to identify where data-races can occur without any use of synchronization and then put that code in a block:

  ```
  __transaction_atomic {
          /* in principle any amount of reads and writes */
  }
  ```

- Too many memory accesses in transactions usually lead to conflicts and transaction-restarts.
- You need to compile with

  ```
  gcc file.c -fgnu-tm -O3
  ```

- Too many memory accesses in transactions usually lead to conflicts and transaction-restarts.
- There is a good chance your program will work at the first attempt.
- Start with studying the `swish.clj` program. The purpose is to see how `ref` variables to array elements can be modified, and how incrementing or decrementing an attribute of an object can be done using `update`.
- Note `recur` used to call the same function recursively, and `first` and `rest` to get the data in a list and the rest of the list, respectively.

- Then make a sequential `preflow.clj` complete.
- Add threads using the same technique as in the swish program. As when using TM for C, your program should work at once.
- All C compilers on Power support the `--help` option but you probably don't want to read all. Some useful flags include:
  - `pgcc -Minfo=all`
  - `xlc -qreport` which will produce a file with suffix `lst`.
- Note that in the `matmul` function, the `c[k][j]` reference will result in very bad cache performance since each iteration of the inner loop is to a different row, which typically is not in the cache. Can you change that in a trivial way (which clang appears to miss)?