Experiments with multitasking and multithreading in LuaTeX

Luigi Scarso

Abstract

We describe a SWIG wrapper to the Pthreads and ZeroMQ C libraries and how they can be used to add multitasking and multithreading features to the Lua interpreter of LuaTeX. Simple examples are shown.

Sommario

Presentiamo un *wrapper* SWIG a Pthreads e le librerie C ZeroMQ e il modo di usarli per aggiungere le proprietà di *multitasking* e *multithreading* all'interprete Lua di LuaTeX. Mostriamo anche dei semplici esempi.

1 Introduction

The average TeX user currently uses a recent TeX distribution on a quite powerful hardware, usually a multicore, Intel-based desktop or a notebook PC. The operating system is very likely (in alphabetical order) Linux, OS X or Windows. Latest releases of all of them support these Intel CPUs, so a natural question is "Can LuaTeX have some gain if it supports these additional processors?" Before answering the obvious way, we will consider two kinds of support. In the first one, that we call *implicit*, the program source code (in our case LuaTeX written in C) is modified by translating, whenever possible, a sequential chunk of code into an equivalent parallel one: e.g., supposing that `a` is an array of 8 integers, the `for` loop

```
for(int i=0;i<8;i++)
  a[i]=2*i;
```

can be executed in parallel with a max speed up $S = 8\times$ if we have at least eight processors.

The programmer usually has to indicate the use of a parallel `for` to the compiler (for example with `pfor`) and let the compiler translate the loop body into parallel code. We can immediately see pros and cons of the internal support: the end user (the TeX user) doesn't have to know any new instruction (i.e., TeX macros are still valid) to gain the support of extra processors but, on the other side, the same user needs a compiler that supports a minimal set of "parallel constructs" which work at least under Linux/OS X/Windows. This last point is the least difficult: OpenMP (*Open MultiProcessing*) is a C library that offers some parallel constructs by `#pragma` directives, so if a compiler doesn't support these directives it's still able to compile the code. For example the aforementioned loop statement can be translated in OpenMP in

```
#pragma omp parallel for
for (i = 0; i < 8; i++)
  a[i]=2*i;
```

OpenMP is supported by the following compilers: GNU `gcc`, Intel `icc` and Microsoft `cl.exe`, while the `LLVM` support is ongoing. OpenMP is a well-established and supported library, with a good community and a rich documentation (see for example Chapman *et al.* (2007)) but it doesn't automatically reveal the internal parallelism of a program. This task requires a huge amount of resources in terms of man-hour and the benefits can be less than expected. Keep considering the `for` loop example already seen: while it's reasonable to expect that each `2*i` operation is performed in parallel, how are memory assignments `a[i]=2*i` performed? They require a shared memory which is, as a consequence, a shared resource requiring a synchronized access. This access can in turn reduce the parallelism.

Even if we think of an architecture with some amount of private memory, many situations require the *communication* of data between different processors and hence again a synchronized access to a shared resource (the bus of communication in this case). Different hardware implementations can show different performances with the same code.

Another important point is the Amdahl's law to calculate the speed up S:

$$S = \frac{T_{\text{seq}}}{T_{\text{par}}} = \frac{1}{(1 - F_e) + F_e/S_e}$$

where T_{seq} and T_{par} are the execution time for the sequential version and for the parallel version of the program, F_e is the fraction of the original time in the sequential execution that can be converted in a parallel one and S_e the speed up that can be obtained if *all* the sequential code can be converted into a parallel one (Hennessy e Patterson, 2012, p. 46). So, for example, if we have eight processors and want a speed up $S = 4$, it might seems reasonable that only 50% of the code need to be rewritten. Amdahl's law says instead that $4 = \dfrac{1}{(1 - F_e) + F_e/8} \Longrightarrow F_e = 6/7 \approx 86\%$ of the code needs to be rewritten. If we just rewrite

50% of the same code, we will have a mere speed up of $S = 1.77$.

The second way to support more than a processor is to *explicitly* delegate the end user to manage concurrent jobs. It's better to explain now that the difference between parallelism and concurrency consists in timings: "a system is said concurrent if it can support two or more actions in progress at the same time. A system is said to be parallel if it can support two or more actions executing simultaneously" (Breshears, 2009, p. 2). According to the previous definition, a system with one processor can be concurrent but it is not parallel; a system with at least two processors can be parallel (and hence concurrent).

It can be a bit surprising but, since all of the aforementioned OSs support background execution, often by means of the system shell, a simple solution for a concurrent TEX is to run in background the needed jobs: the `shell_escape` environment variable must be enabled and some macros have probably to be written to encapsulate the specific call in the system shell. The problem is the communication of data between different jobs, that can be inefficient or even impossible, and the fact that there is a poor control over the job themselves (for example, it can be impossible to kill a background job).

In this paper we will consider how to use LUATEX to explicitly manage parallel/concurrent jobs, so we will not consider the OpenMP library. In the next sections we will first introduce a minimal terminology, then we will present the used tools, then we will conclude showing some simple examples not specifically tailored for LUATEX.

2 Processes, threads and processors

At the very beginning of the computer's age, a typical computer had an expensive CPU and central memory, both limited in speed and size. This means that the first Operating Systems (*OS*) were not too complex; even the work organization had to be quite elementary: each job had to be executed in a serial way (*batch processing*) almost without user interaction. This will be the optimal organization (i.e., it maximizes CPU and memory usage) if each job has no interaction with Input/Output (*I/O*) devices; if a job had a long session with a slow I/O (like a long printing) the CPU would have to wait until the end of the session while it could be profitably used to serve one of the waiting jobs. With the increasing power of CPUs and memory size, the next step was the creation of a primitive OS to manage more than a job still executing a single job at time: if a job is waiting the end of an I/O operation, the OS saves (a representation of) this job in the memory and gets the next job; when an operation ends, the associated job has to wait until the processor is again assigned to it. This technique is called *multiprogramming*.

Before going on, it is better to remind some terms. A *programming language* is a *formal language* that has a machine as target; a *computer* programming language has a computer as target. An alphanumerical string is called a *program* in a programming language if 1) it is written according to the syntax and the semantic of the given programming language and 2) it correctly translates an algorithm. The machine (i.e. a computer) running the program progressively reveals the underneath semantic *executing* program statements, that we can consider more elementary pieces of the program itself. The execution of a program requires that the OS creates and fills the appropriate data structures in the memory so that the CPU can find the machine instructions to execute. When the execution ends, data structures are erased, freeing the memory for another execution. Data along with CPU registers form the *task* or *process* associated to the program. The term process is generally used in the context of the data structures for a specific OS, while task is used in broader sense. Anyway, both of them indicate a *dynamic* entity, i.e an activity that has a starting and a running time and that, under certain circumstances, can be stopped and restarted.

An OS is said *multitasking* (seldom *multiprocessing*) if it supports more than one task at the same time. There are two kinds of multitasking: *cooperative*, where the task itself returns control to the OS or releases resources, and *preemptive*, where the OS will manage the execution of tasks, stopping one of them after a fixed amount of time (*quantum* or *time slice*), and resumes the run of another task. There is a difference between a multiprogramming and a multitasking OS. The latter is designed to use a single CPU and memory that are fast and large so that interactions with the user is no more negligible. The OS will indeed manage system and users processes to give all an adequate amount of CPU time (*fairness*) and will guarantee an overall good responsiveness. On the other side, like in the multiprogramming, each process still sees the computer as a dedicated CPU with a memory of contiguous addresses starting from zero and ending to some limit (*flat memory model*), eventually higher than the physical memory actually available in the computer, and the OS takes care to keep separate each process space. If a process tries to access an invalid address it will be terminated (*killed*) by the OS. This is a mandatory feature for a multitasking OS, but it comes with a price of big data structures. The time to create a process, or to clone an existing one, is long, as is the switch from a process to another one (*process context switch*). Another complication is the communication between processes (*InterProcess Communications* or IPC) that involves the

OS due to the need to translate memory addresses between processes. A solution to these problems is to allow the existence of "processes" inside the same address space, the *threads*. A process is hence a collection of one or more threads and the key point is that the context switch between threads is faster than between processes (even ten times, see Kerrisk (2010, p. 610) for Linux, but similar figures are also valid for Windows). On the other side, if a thread tries to access a wrong memory address, the entire process and its related threads are killed; the same stands when a thread exits (i.e. signals to the OS that its process is terminated) all the other threads also are killed in case they have finished their run.

At this point we have to consider that a current common desktop computer has more than one processor. More precisely, a *motherboard* can have one, two or four *sockets* hosting a *Central Processing Unit* (CPU), and each CPU can have from two to four *cores*. Each core can manage a different process, and in the most common hardware architecture, the SMP (*Symmetric MultiProcessor*), all cores are peers. A process running in a core can be stopped, moved to another core and resumed. Furthermore, the Intel "Hyper-Threading Technology" (HTT) splits each core into two logic CPUs with shared L1 cache. This is a proprietary implementation of the *Simultaneous MultiThreading* (SMT) technique that maps a thread into a virtual CPU to maximize the use of a core (the rationale is that two threads of the same process share the same address space so the cache L1 and L2 caches have better chances to already have the needed instructions and data). The HTT is a hardware feature and need support from the OS. As examples, computers used for this paper have a K55V single socket motherboard by ASUSTeK COMPUTER INC. hosting an Intel© CORE™i7-3610QM at 2.30GHz with 4 cores which implements the HTT, offering 8 virtual CPUs, and a T101MT single socket motherboard, from ASUSTeK COMPUTER INC. as well, equipped with an Intel© ATOM™ N450 at 1.66GHz with 1 core that still implements the HTT, offering 2 virtual CPUs. The first one is in a notebook running Linux and the second one in a netbook running Microsoft Windows 7 Home Premium[1] and both the CPUs are soldered on the motherboard (so practically there is no socket). Both OSs support SMP and SMT (see Kerrisk (2010) for Linux and Russinovich *et al.* (2012a) and Russinovich *et al.* (2012b) for Windows 7), but under Windows the SMP is not enabled because there is only one core. As we can see, the single processor era is ended.

1. On Linux this information is available using the `dmidecode` command; on Windows, using the `System Information` command

3 Supporting threads and processes under LuaTEX

In the first section we have seen that OpenMP is a widely used library to expose the implicit parallelisms. The question we would like to answer is "Is there something similar for explicit parallelism/concurrence in TEX?" We can consider the following facts:

- TEX has no construct for concurrence but most, if not all, current modern TEX implementations have the `\write18` macro that can execute a system command, so it can be used to launch a program in background;
- Lua (and hence luaTEX) *supports* cooperative multithreading by `coroutines` (Ierusalimschy, 2013, ch. 9) but this doesn't mean that a Lua `thread` can be scheduled by the underlying OS as a user process/thread. Anyway Lua (and luaTEX) can call an external C library when a suitable wrapper is present, and this one *can* create a user process/thread;
- despite Linux, OS X and Windows are different OSs, they share the same concept of process and thread[2] and they are mostly implemented in C, with (usually) Assembly code for drivers and C++ code for some abstract constructs (OS X also uses Objective C). Standard C has no support for processes/threads (only the latest C++11 standard has the new `std::thread` class).

The problem to come is to find a library that at least provides support for threads. Under UNIX©, the most important library is POSIX Threads (Pthreads), which is part of IEEE standard 1003.1 (see `http://pubs.opengroup.org/onlinepubs/9699919799`). Linux has an almost full support for Pthreads and the API are described in Kerrisk (2010, ch. 28–33); OS X fully supports Pthreads (see `http://www.opengroup.org/openbrand/register/brand3591.htm`) and, even if Pthreads are not directly supported in Windows[3], the Pthreads win32 project at `https://sourceware.org/pthreads-win32/` is a ten-years-old project, still active, which offers an almost complete implementation of the API. The other important point is that the API is specified by a set of C header files so it looks reasonable to try to build a Lua wrapper for luaTEX.

3.1 A SWIG wrapper for Pthreads

One of the key points of Lua design is to facilitate interfacing with external libraries with a complete

2. Windows has also a variant of threads, called *fiber*, similar to Lua `coroutines`.

3. Windows has a partial support for POSIX.1 as subsystem, but not for Pthreads.

API. SWIG, the *Simplified Wrapper and Interface Generator*, is a program that assists the developer in building a wrapper and can automatize many aspects. Supposing that header files that list the API are in the `pthread` subfolder of the current folder, a typical workflow consists in preparing an *interface* file `core.i` like this:

```
/*   core.i */
%module core
%{
#include "pthread/sched.h"
#include "pthread/semaphore.h"
#include "pthread/pthread.h"
%}
%include "pthread/sched.h";
%include "pthread/semaphore.h"
%include "pthread/pthread.h"
```

and let `swig` produce the C source of the wrapper, `core_wrap.c` with

```
swig -importall -I/usr/include
 -I/usr/lib/gcc/x86_64-linux-gnu/
    4.7/include/
 -lua core.i
```

Then the source of the wrapper is compiled and linked against the Pthreads library:

```
gcc -fpic -I./pthread -pthread \
     -c core_wrap.c -o core_wrap.o
gcc -Wall -shared -pthread core_wrap.o \
     -llua5.2 -lpthread  -o core.so
```

That's all: the C API are now available to Lua (and luaTEX):

```
local _core= package.loadlib("./core.so",
                          "luaopen_core")
if not(_core) then
  print("error loading _core")
  return 1
end
local pthread = _core()
for k,v in pairs(pthread) do
  print(k,v)
end
```

A complete interface file (for Linux) is given in fig. 2 and 3, but it is better to discuss some aspects of threads, C and Lua before commenting the file.

One of the main problem of processes and threads is the management of a shared resource, which almost always requires a synchronized access. We can think of a shared resource like a cross between fours roads: without synchronized semaphores a crash is almost secure if nobody takes care to respect the rules. With threads, the memory is a shared resource because, as we have seen in the Introduction, each thread inside a process shares the same address space. This is a key point: it means that in C it is possible to implement threads with functions. Looking at fig. 3 we can see at line `87` that a thread is started by a call to `worker(arg)` where `worker` is in fig. 2, line `25`. A call to `pthread_create` immediately starts a thread `worker` and returns, so it is available to start other threads and the problems raise because all of the variables are inside the same address space, so we must be sure that each thread doesn't interfere with others threads. Now, in C variables are of three types (see Reese (2013, p. 2)):

static/global: A static variable has its scope inside the function where it is declared, but its lifetime is that of the process. A global variable has its scope inside the program and the same lifetime of the static variable. For this reason static and global variables need the special treatment assured by threads;

automatic: An automatic variable has its scope inside the function that declared it and its lifetime corresponds to the duration of the function. The C runtime *automatically* manages these variables so there are no problems with threads. Local variables and parameters of a function are automatic variables so they are *thread safe*;

dynamic: it is a variable created with `malloc` or equivalent function. This is usually a pointer that can be passed (by value) to a function, and hence is a sort of global access to the memory addressed by the pointer. Its lifetime ends when and whether the memory is freed. This means that it can lead to *memory leak* when the memory is never released. Threads must carefully manage a pointer of this type.

luaTEX has several global/static variables and a thread-safe version of the program implies a rewriting of a consistent part of the code; it is more convenient considering only the Lua interpreter because, as explained in Ierusalimschy (2013, p. 251), each Lua function receives a pointer to `lua_State` and uses exclusively this state, and this "implementation makes Lua reentrant and ready to be used in multithreaded code". We can hence follow the idea of Ierusalimschy (2013, sec. 31.2): each thread (the `worker` function) creates its own Lua state (in fig. 2, line `27`) and this interpreter executes a chunk of Lua code in (fig. 2, line `48`). As we can see in fig. 3, there are two dynamic variables: `code_clone` (line `70`), that has a copy of the chunk so there is no interference with the `lua_State` of luaTEX, and `arg` (line `75`) which is `worker` argument. It's not possible to free these variables in `swiglib_pthread_create`, because this function can return *before* the `worker` is started, so this last one would not see valid data. This means that `worker` has to free these variables after their use.

We can see an example in fig. 4, where three threads are created and all of them run the code

in code_base (line 17). As expected, the output in fig. 5 shows the printed lines output without a particular order. The thread workers are *joinable*, thus the master can synchronize itself to their termination (lines 46 to 48 in fig 4).

We can make the following considerations:

1. in the C code there is no use of any primitive to synchronize the Pthreads API (mutex, semaphore) because there are not any static/global variables and dynamic variables are not shared (they are managed by the worker). In spite of this, **the code is not thread-safe** due the void pointer data in the struct thread_worker_arg (fig. 2, line 23);

2. it seems that also in the Lua code there are not any shared resources, but **this is not true**. Indeed in fig. 4, line 27 there is a call to os.date() which can call the system function localtime(); this one is not thread-safe (see http://pubs.opengroup.org/onlinepubs/009695399/, subsec. 2.9.1) There is a thread-safe version of localtime, localtime_r, used by Lua 5.2.2 (and hence LuaTeX 0.76.0) if it is available at compile-time (i.e., in a POSIX system, but it works also under Linux even if it's not fully POSIX-compliant). It is *not used* in Lua 5.1.4 (and hence LuaTeX until TeX Live 2012). luatex.exe for Windows is linked against msvcrt.dll, *which is thread safe.*

Considering observation 1. we can say that the void pointer data is a way to pass user data to a thread. We show it with an example when the user data is an int. First we have to create a pointer to our data; this is done in the interface with the line %pointer_functions(int , int_p); (fig. 2, line 15). This means that on the Lua side there are now those functions to manage this type of user data:

```
local data = pthread.new_int_p()
pthread.int_p_assign(data,99999)
```

We can pass data to the thread:

```
pthread.swiglib_pthread_create(
   t0,attr,-1,
   code0,string.len(code0),
   data)
```

In fig. 2, lines 42 to 47 data is stored as a *lightuserdata* in the swiglib_pthread_data Lua variable and we can read it as shown in fig. 1.

The main advantage of the lightuserdata is that it is not managed by the garbage collector of Lua in the thread, so there is no risk for the pointed memory to be erased, but on the other side it is the caller's responsibility to protect and manage this resource because the worker doesn't take any action. In this sense worker is not thread-safe. Moreover, we must supply two other functions: pthread.lightuserdata_touserdata_void_p and pthread.void_p_to_int_p (in fig. 2, lines 91 and 108) to translate a lightuserdata to an int. If we don't need to pass data we can set data=nil as in fig. 4 and there is no risk because swiglib_pthread_data is also set to nil. This technique is general, because with SWIG we can define our data as a struct (as in fig. 2, line 20) and then provide the pointer functions and the lightuserdata conversion functions.

According to point 2., we can mitigate the problem selecting the module to load. The auxlibs of arg (which is passed by value, thus it's thread-safe) it's a bit mask to select which module to load into the Lua state of the thread (in fig. 2, lines 41 to 43): -1 loads the entire module. But in general we should check that the system functions used by Lua are thread-safe.

As we can see, Pthreads is a low-level library and its use requires some knowledge that is probably not so common to the average TeX user. In the next subsection we will see another library do the deal with concurrency in a more abstract way.

3.2 A SWIG wrapper for ZeroMQ

ZeroMQ is a C library for Message passing. In a broader sense, message passing concerns the exchange of messages between threads, be they in the same process, in different processes on the same instance of an OS, in different OSs on different machines. The key concept is a generalization of the UNIX© socket, the *zmq socket*, which has several *types*, each one defining a *message pattern*:

Request-reply: connects a set of clients to a set of services.

Publish-subscribe: connects a set of publishers to a set of subscribers. This is a data distribution pattern, which means that the reference data is published in a service bus and only new subscribers need to be added to the service bus; there is no need to change the service to serve new targets.

Push-pull: connects nodes in a fan-out / fan-in pattern that can have multiple steps and loops. This is a parallel task distribution and collection pattern.

Exclusive pair: connects two sockets in an exclusive pair. This is a low-level pattern for specific, advanced use cases and still experimental.

(please refer to http://en.wikipedia.org/wiki/Messaging_pattern.)

ZeroMQ is a mid-level abstraction library; it can be used to replace some functionality of Pthreads; it has not its own low-level mechanisms of synchronization but, on the other side, it has not any built-in server program (like http or ftp servers) — it can be used to build custom versions of these programs. For example, it is currently used by

```
1  local code_base =
2  [=[
3    local th = "%s"
4    local _core= package.loadlib("./core.so","luaopen_core")
5    if not(_core) then print(th .. ":error loading _core") return 1 end
6    local pthread = _core()
7    if swiglib_pthread_data~=nil then
8      local d1 = pthread.lightuserdata_touserdata_void_p(swiglib_pthread_data)
9      print(th.." data value=",pthread.int_p_value( pthread.void_p_to_int_p(d1) ) )
10   end
11   print(th .. " end")
12 ]=]
```

FIGURE 1: Passing a user data to a thread (no thread safe)

CERN to update their Controls Middleware system software previously based on the CORBA middleware (DWORAK *et al.*, 2012)[4]. The authoritative source of documentation is the site `http://zguide.zeromq.org` and the new book HINTJENS (2013).

A SWIG wrapper for ZeroMQ is straight:

```
%module core
%{
#include "zeromq/zmq.h"
%}
%include "zeromq/zmq_utils.h" ;
%include "zeromq/zmq.h" ;
```

Building the wrapper is also simple (here shown for Linux), and it needs the Pthreads library:

```
LUAINC52=/usr/include/lua5.2
LIBS="-lpthread -lzmq"
CFLAGS="-g  -O2 -Wall -I./zeromq \
           -pthread"
swig  -lua core.i
rm -vf core_wrap.o
gcc   -O2 -fpic -I./zeromq -I$LUAINC52 \
      -c core_wrap.c  -o core_wrap.o
rm -vf core.so
gcc -Wall -shared -O2 \
    -Wl,-rpath,'$ORIGIN/.'\
    $CFLAGS \
    core_wrap.o  \
    -llua5.2 $LIBS \
    -o core.so
```

In fig. 6 we show a push-pull example: the main process opens a `ZMQ_PULL` zmq socket connected to the TCP port 5555 and *launches* `max_workers` *processes in background* (line 20). After that it "pulls" the results with a `receive` (fig. 6, line 32). Lines 24 to 49 ensure that all workers are served[5]. Each worker shares the same file shown in fig. 7: it opens a `ZMQ_PUSH` zmq socket connected to the same TCP port and "pushes" data with a `send` (fig. 7, line 24). The mentioned data is an array of char, managed by the external library `helpers` — a simple SWIG wrapper to `char[]`:

```
%module core
%include "carrays.i"
%array_functions(char, char_array);
```

The key point here is that we don't use threads but processes, so there is no need for synchronized accesses; on the other side the way we run a process in background now depends on the underlying OS: under Windows we have to use the following string for `os.execute`:

```
"start 'worker' /b luatex.exe
  'ex008-PUSH_PULL-worker.lua' '%s' 2>&1"
```

ZeroMQ also manages message passing between threads with the protocol `inproc://<name>`. Fig. 8 and fig. 9 show the same push-pull pattern with threads. Running times are as expected: the process version lasts

```
$ time ./luatex ex008-PUSH_PULL-process.lua \
   >/dev/null
real 0m12.120s
user 0m0.196s
sys 0m0.852s
```

while the threaded version lasts

```
$ time ./luatex ex008-PUSH_PULL-joined.lua \
   >/dev/null
real 0m0.712s
user 0m0.064s
sys 0m4.952s
```

4. "We are migrating an infrastructure with 3500 active servers and 1500 active clients from CORBA to ZMQ. We only provide the library, our users implement the clients/servers on top of it and deploy where/how they want. We also have many combinations of hardware/OS: low level front-ends, middle tier servers and workstations." (see `http://comments.gmane.org/gmane.network.zeromq.devel/18437`).

5. It's a very primitive mechanism for synchronization: production code must use more robust solutions as explained in HINTJENS (2013).

i.e. approximately 17 faster. The approach of ZeroMQ is clear: *don't share state* and hence there is no need for synchronization to access shared resource. In fact the only object shared is the `context` which is declared as thread safe (on the other hand, a zmq socket is not thread safe so it must not be shared).

3.3 Considerations for integration in LuaTEX

The current implementations of LuaTEX, pdfTEX and XeTEX support at least Linux, OS X and Windows, so it's almost a mandatory requirement that an extension library also works at least under the same OSs. We have made our experiments on Linux 64bit and Windows Home premium 32bit and we have still to do tests under Windows 64bit. Due to the lack of a machine with the latest OS X, we have not made tests with this OS, but we are confident that there we would not have serious problems because it is a POSIX system and the clang compiler is a mature project, at least for the C language. An important problem is where to put these so/dll libraries without cluttering the OS: for example, under Linux, passing `-Wl,-rpath,'$ORIGIN/.'` to the linker has as a consequence that the libraries can be found starting from the directory of the local application, but a similar switch is not available under Windows which has different rules. LuaTEX has the Lua file system module `lfs` from `http://keplerproject.github.io/luafilesystem/` but it's not usable with a worker because it has a separated state. A solution is then to load a `lfs` module into the worker, so we can switch from the current directory to that one having the module to load, as in the following chunk:

```
_core = package.loadlib("lfs/lfs.dll",
                        "luaopen_lfs")
if not(_core) then print("error lfs")
  return 1 end
local lfs = _core()
local current_dir = lfs.currentdir()
lfs.chdir("./zeromq")
local _core= package.loadlib(
               "./zeromq/core.dll",
               "luaopen_core")
if not(_core) then print("error zmq")
  return 1 end
local zmq = _core()
lfs.chdir(current_dir)
```

The Swiglib project `http://www.luatex.org/swiglib.html` tries to address these and other aspects, as for example a generic `helper` module.

While it is clear that Pthreads and ZeroMQ open new possibilities to efficiently manage different data sources for typesetting, the other aspect — probably the most interesting one — of interacting with LuaTEX to add concurrency to the task of typesetting still remains to explore, but we can delineate two possibilities:

process: run several instances of LuaTEX coordinate by ZeroMQ, typically with a `tcp` connection. This could be useful for example in processing a book if some chapters are mutually independent;

thread: inside a single instance of LuaTEX use worker threads that cooperate with the Lua state of LuaTEX. For example a two columns layout could be rendered in a pipeline by two different threads, or n threads could be used in parallel to build a vbox with different parameters and then choose the best one.

The process possibility means that we should have identical instances, but this is usually not a problem (just uses the same version of LuaTEX and format for each instance); of course the runs cannot put files in the same folder because each instance would interfere with the others. The problem is to resolve the implicit dependencies (as, for example, page and chapter numbers), and the synchronization of the common parts, as for example the index.

The thread possibility looks more complex. Communicating with the Lua internal state of LuaTEX is complicated because there is not any API — which is reasonable: it is a program. A more feasible approach is a pure Lua implementation of some parts of LuaTEX, as for example the line breaking algorithm. In this case the problem is how to exchange data: `void *data` is not thread-safe, so it must be carefully managed.

4 Conclusions

The implicit parallelism offered by libraries like OpenMP can give LuaTEX an effective support for multiprocessors, but simple estimations indicate that the amount of code to rewrite is extremely huge so that any potential advantage is lost.

Explicit parallelism/concurrency by external libraries is more convenient for an existing program like LuaTEX that can take advantage of the very flexible Lua interpreter. The Pthreads and ZeroMQ libraries cover the "dual" aspect of processes vs threads giving a good coverage of the subject and we have shown some simple examples working with the Lua interpreter of LuaTEX. We think that these examples justify more investigations explicitly focused on typesetting problems, especially in the area of adding parallelism to LuaTEX using threads. We are open to suggestions and critiques to get the best solution.

The code is hosted in the SVN server of the Swiglib project at `https://foundry.supelec.fr/scm/viewvc.php/trunk/experimental/?root=swiglib`.

References

Breshears, C. (2009). *The Art of Concurrency: A Thread Monkey's Guide to Writing Parallel Applications*. O'Reilly Media, Inc.

Chapman, B., Jost, G. e Pas, R. v. d. (2007). *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*. The MIT Press.

Dworak, A., Ehm, F., Charrue, P. e Sliwinski, W. (2012). «The new CERN Controls Middleware». *Journal of Physics: Conference Series*, **396** (1), p. 012 017. URL `http://stacks.iop.org/1742-6596/396/i=1/a=012017`.

Hennessy, J. L. e Patterson, D. A. (2012). *Computer Architecture — A Quantitative Approach*. Morgan Kaufmann, 5ª edizione.

Hintjens, P. (2013). *Code Connected Volume 1: Learning ZeroMQ*. Code Connected Series. CreateSpace Independent Publishing Platform. URL `http://books.google.it/books?id=hY6olQEACAAJ`.

Ierusalimschy, R. (2013). *Programming in Lua, Third Edition*. Lua.Org.

Kerrisk, M. (2010). *The Linux Programming Interface: A Linux and UNIX System Programming Handbook*. No Starch Press, San Francisco, CA, USA, 1ª edizione.

Reese, R. (2013). *Understanding and Using C Pointers*. O'Reilly Media, Incorporated. URL `http://books.google.it/books?id=7dOwkQEACAAJ`.

Russinovich, M. E., Solomon, D. A. e Ionescu, A. (2012a). *Windows Internals, Part 1: Covering Windows Server 2008 R2 and Windows 7*. Microsoft Press, 6ª edizione.

— (2012b). *Windows Internals, Part 2: Covering Windows Server 2008 R2 and Windows 7 (Windows Internals)*. Microsoft Press.

▷ Luigi Scarso
`luigi dot scarso at gmail dot com`

```
%module core
%include "carrays.i"; %include "cpointer.i"; %include "constraints.i";
%include "cmalloc.i"; %include "lua_fnptr.i";
%{
#include "pthread/sched.h"
#include "pthread/semaphore.h"
#include "pthread/pthread.h"
%}
%ignore worker; %ignore thread_worker_arg;
%include "pthread/sched.h";
%include "pthread/semaphore.h"
%include "pthread/pthread.h"
%pointer_functions(pthread_attr_t,pthread_attr_t_p);
%pointer_functions(pthread_t,pthread_t_p);
%pointer_functions(int , int_p);
%array_functions(pthread_attr_t *,pthread_attr_t_p_array);
%array_functions(void *,void_p_array);
%pointer_cast(void *,int *,void_p_to_int_p);
%pointer_cast(int *,void *,int_p_to_void_p);
%inline %{
struct thread_worker_arg{
  char *code;          int auxlibs;
  size_t is_joinable;  void *data;
};
void *worker(void *thread_arg){
  struct thread_worker_arg *arg = (struct thread_worker_arg *) thread_arg;
  lua_State *L = luaL_newstate();
  char *code = arg->code; int auxlibs = arg->auxlibs;
  int res ;  int *retval;
  if (auxlibs<0){
    luaL_openlibs(L);
  } else if (auxlibs==0) {
    luaopen_base(L);
  } else {
    luaopen_base(L);
    if (auxlibs & (1<<0)) luaopen_coroutine(L);  if (auxlibs & (1<<1)) luaopen_table(L);
    if (auxlibs & (1<<2)) luaopen_io(L);     if (auxlibs & (1<<3)) luaopen_os(L);
    if (auxlibs & (1<<4)) luaopen_string(L); if (auxlibs & (1<<5)) luaopen_bit32(L);
    if (auxlibs & (1<<6)) luaopen_math(L);   if (auxlibs & (1<<7)) luaopen_debug(L);
    if (auxlibs & (1<<8)) luaopen_package(L);
  }
  if (arg->data==NULL) {
    lua_pushnil(L);
  } else {
    lua_pushlightuserdata (L, arg->data);
  }
  lua_setglobal(L, "swiglib_pthread_data");
  res = luaL_loadstring(L,code);
  if (res==0)
    res = lua_pcall(L,0,0,0);
  /* is this th joinable ?*/
  if (arg->is_joinable==PTHREAD_CREATE_JOINABLE){
    /* main thread must free retval */
    retval = malloc(sizeof(int));
    *retval = res;
  } else {
    retval=NULL;
  }
  free(arg->code);  free(arg);  lua_close(L);
  return (void *) retval;
}
%}
```

FIGURE 2: Interface file `core.i` (1 of 2)

```
#include <string.h>
int swiglib_pthread_create(pthread_t *thread,const pthread_attr_t *attr,
                           int auxlibs, const char *code, int code_len,
                           void *data) {
    int res; char *code_clone; int attr_value;
    if (code_len<0)  return -1000;
    if (code_len==0) return -1001;
    code_clone = malloc((sizeof(char)*code_len)+1);
    if (code_clone==NULL) return -1002;
    code_clone[code_len]=0;
    code_clone = strncpy(code_clone,code,code_len);
    if (code_clone==NULL) return -1003;
    struct thread_worker_arg *arg = malloc(sizeof(struct thread_worker_arg));/* NO ANSI C !*/
    if (arg==NULL) return -1005;
    /*worker(s) must free them !! */
    arg->code = code_clone;
    arg->auxlibs=auxlibs;
    arg->data = data;
    if (attr==NULL){
      arg->is_joinable=PTHREAD_CREATE_JOINABLE;
    } else {
      if (pthread_attr_getdetachstate(attr, &attr_value)!=0) return -1006;
      arg->is_joinable=attr_value;
    }
    res=pthread_create(thread, attr, worker, arg);
    return res;
}
%}
%native(lightuserdata_touserdata_int_p)
  static int native_lightuserdata_touserdata_int_p(lua_State*L);
%{int native_lightuserdata_touserdata_int_p(lua_State*L){
  int SWIG_arg = 0;
  int *result = 0 ;
  SWIG_check_num_args("lightuserdata_touserdata_int_p",1,1);
  if(!lua_islightuserdata(L,1))
    SWIG_fail_arg("lightuserdata_touserdata_int_p",1,"lightuserdata int *");
  result = (int *)lua_touserdata(L,1);
  SWIG_NewPointerObj(L,result,SWIGTYPE_p_int,0); SWIG_arg++;
  return SWIG_arg;
  if(0) SWIG_fail;
fail:
  lua_error(L);
  return SWIG_arg;
}
%}
%native(lightuserdata_touserdata_void_p)
   static int native_lightuserdata_touserdata_void_p(lua_State*L);
%{int native_lightuserdata_touserdata_void_p(lua_State*L){
  int SWIG_arg = 0;
  void *result = 0 ;
  SWIG_check_num_args("lightuserdata_touserdata_void_p",1,1);
  if(!lua_islightuserdata(L,1))
   SWIG_fail_arg("lightuserdata_touserdata_void_p",1,"lightuserdata void *");
  result = (void *)lua_touserdata(L,1);
  SWIG_NewPointerObj(L,result,SWIGTYPE_p_void,0); SWIG_arg++;
  return SWIG_arg;
  if(0) SWIG_fail;
fail:
  lua_error(L);
  return SWIG_arg;
}
%}
```

FIGURE 3: Interface file `core.i` (2 of 2)

```
local _core= package.loadlib("./core.so","luaopen_core")
if not(_core) then print("error loading _core") os.exit(1) end
local clock = os.clock
function sleep(n)  -- seconds
  local t0 = clock()
  while clock() - t0 <= n do end
end
local pthread = _core()
attr  = pthread.new_pthread_attr_t_p()
pthread.pthread_attr_init(attr)
pthread.pthread_attr_setdetachstate(attr, pthread.PTHREAD_CREATE_JOINABLE)

t0    = pthread.new_pthread_t_p()
t1    = pthread.new_pthread_t_p()
t2    = pthread.new_pthread_t_p()

local code_base =
[=[
  local clock = os.clock
  function sleep(n)  -- seconds
   local t0 = clock()
   while clock() - t0 <= n do end
  end
  local s,s1
  local th="%s"
  for i=1,%s do
   local s=os.date()
   local s1=th.."<"..tostring(s).." "..tostring(i)..">"
   io.write(s1,"\n")
   sleep(%s)
  end
]=]


local code0=string.format(code_base,"1","10","0.4")
local code1=string.format(code_base,"2","10","0.3")
local code2=string.format(code_base,"3","10","0.4")
local data = nil
local rc

rc = pthread.swiglib_pthread_create(t0,attr,-1,code0,string.len(code0),nil)
rc = pthread.swiglib_pthread_create(t1,attr,-1,code1,string.len(code1),nil)
rc = pthread.swiglib_pthread_create(t2,attr,-1,code2,string.len(code2),nil)
pthread.pthread_attr_destroy(attr);

pthread.pthread_join(pthread.pthread_t_p_value(t0),nil)
pthread.pthread_join(pthread.pthread_t_p_value(t1),nil)
pthread.pthread_join(pthread.pthread_t_p_value(t2),nil)

print("end")
```

FIGURE 4: Test running 3 threads joined (1 of 2)

```
2<Fri Sep 13 18:07:54 2013 1>
3<Fri Sep 13 18:07:54 2013 1>
1<Fri Sep 13 18:07:54 2013 1>
2<Fri Sep 13 18:07:54 2013 2>
3<Fri Sep 13 18:07:54 2013 2>
1<Fri Sep 13 18:07:54 2013 2>
2<Fri Sep 13 18:07:54 2013 3>
3<Fri Sep 13 18:07:54 2013 3>1<Fri Sep 13 18:07:54 2013 3>

2<Fri Sep 13 18:07:54 2013 4>
1<Fri Sep 13 18:07:54 2013 4>
3<Fri Sep 13 18:07:54 2013 4>
2<Fri Sep 13 18:07:54 2013 5>
2<Fri Sep 13 18:07:54 2013 6>
3<Fri Sep 13 18:07:54 2013 5>
1<Fri Sep 13 18:07:54 2013 5>
2<Fri Sep 13 18:07:54 2013 7>
3<Fri Sep 13 18:07:54 2013 6>
1<Fri Sep 13 18:07:54 2013 6>
2<Fri Sep 13 18:07:55 2013 8>
3<Fri Sep 13 18:07:55 2013 7>
1<Fri Sep 13 18:07:55 2013 7>
2<Fri Sep 13 18:07:55 2013 9>
2<Fri Sep 13 18:07:55 2013 10>
3<Fri Sep 13 18:07:55 2013 8>
1<Fri Sep 13 18:07:55 2013 8>
3<Fri Sep 13 18:07:55 2013 9>
1<Fri Sep 13 18:07:55 2013 9>
3<Fri Sep 13 18:07:55 2013 10>
1<Fri Sep 13 18:07:55 2013 10>
end
################################################################
3<Fri Sep 13 18:07:55 2013 1>
1<Fri Sep 13 18:07:55 2013 1>
2<Fri Sep 13 18:07:55 2013 1>
2<Fri Sep 13 18:07:55 2013 2>
3<Fri Sep 13 18:07:55 2013 2>
1<Fri Sep 13 18:07:55 2013 2>
2<Fri Sep 13 18:07:56 2013 3>
1<Fri Sep 13 18:07:56 2013 3>
3<Fri Sep 13 18:07:56 2013 3>
2<Fri Sep 13 18:07:56 2013 4>
1<Fri Sep 13 18:07:56 2013 4>
3<Fri Sep 13 18:07:56 2013 4>
2<Fri Sep 13 18:07:56 2013 5>
2<Fri Sep 13 18:07:56 2013 6>
1<Fri Sep 13 18:07:56 2013 5>
3<Fri Sep 13 18:07:56 2013 5>
2<Fri Sep 13 18:07:56 2013 7>
3<Fri Sep 13 18:07:56 2013 6>1<Fri Sep 13 18:07:56 2013 6>

2<Fri Sep 13 18:07:56 2013 8>
1<Fri Sep 13 18:07:56 2013 7>
3<Fri Sep 13 18:07:56 2013 7>
2<Fri Sep 13 18:07:56 2013 9>
2<Fri Sep 13 18:07:56 2013 10>
3<Fri Sep 13 18:07:56 2013 8>
1<Fri Sep 13 18:07:56 2013 8>
1<Fri Sep 13 18:07:56 2013 9>
3<Fri Sep 13 18:07:56 2013 9>
3<Fri Sep 13 18:07:57 2013 10>
1<Fri Sep 13 18:07:57 2013 10>
end
```

FIGURE 5: Test running 3 threads joined, two outputs (2 of 2)

```
print("PULL BEGIN "..os.date())
local _core= package.loadlib("./zeromq/core.so","luaopen_core")
if not(_core) then print(th .. ":error zmq") return 1 end
local zmq = _core()
local f = package.loadlib("helpers/core.so","luaopen_core")
if not(f) then print("Error on loading helpers" ) return 1 end
local helpers = f()
local clock = os.clock
function sleep(n)  -- seconds
  local t0 = clock()
  while clock() - t0 <= n do end
end
local max_workers = 10
-- Socket to receive  messages on
local context  = zmq.zmq_ctx_new();
local recvfrom = zmq.zmq_socket(context, zmq.ZMQ_PULL);
local rc       = zmq.zmq_bind(recvfrom, "tcp://*:5555");
if (rc~=0) then print("error on bind tcp://workers") return 1 end
for k=1,max_workers do
 os.execute(string.format("lua5.2 'ex008-PUSH_PULL-worker.lua' '%s'  2>&1 &",k))
end
local c={}
local i=max_workers+3 -- uhu a sign that push-pull is  wrong
while (i>0) do
  local size = -1
  local bufsize = 256
  local _buffer     = helpers.new_char_array(bufsize)
  helpers.char_array_setitem(_buffer,bufsize-1,"\0")
  local buffer = helpers.char_p_to_void_p(_buffer)
  --msg_size doesn't include the trailing zero "\0"
  msg_size=bufsize-1
  size = zmq.zmq_recv(recvfrom, buffer, msg_size,0);
  local data=""
  if size==-1 then
    data=""
  else
    if size >(bufsize)  then
      size=bufsize-1
    end
    buffer = helpers.void_p_to_char_p(buffer)
    helpers.char_array_setitem(buffer,size,"\0")
    local t={}
    for i=0,size-1 do t[#t+1]=helpers.char_array_getitem(buffer,i) end
    data=table.concat(t)
  end
  c[#c+1]=string.format("SINK: seen %s size=%s data=%s,#data=%s",i-3,size,data,#data)
  i=i-1
  if i==3 then sleep(1) ; i=0 end
end
for k=1,#c do print(c[k]) end
zmq.zmq_close(recvfrom);
zmq.zmq_ctx_destroy(context);
print("PULL END "..os.date())
```

Figure 6: Push-pull: the code of the pull process (1 of 2)

```
local th = tostring(arg[1])
local f = package.loadlib("helpers/core.so","luaopen_core")
if not(f) then print("Error on loading helpers" ) return 1 end
local helpers = f()
local _core= package.loadlib("./zeromq/core.so","luaopen_core")
if not(_core) then print(th .. ":error loading _core") return 1 end
local zmq = _core()
local clock = os.clock
function sleep(n)  -- seconds
 local t0 = clock()
 while clock() - t0 <= n do end
end
local context = zmq.zmq_ctx_new();
local sendto = zmq.zmq_socket(context, zmq.ZMQ_PUSH);
local rc     = zmq.zmq_connect(sendto, "tcp://localhost:5555");
if (rc~=0) then print(th.." worker:error on connect tcp://localhost") return 1 end
local msg = "123456_"..th
local bufsize= #msg+1
local zmq_msg= helpers.new_char_array(bufsize)
helpers.char_array_setitem(zmq_msg,bufsize-1,"\0")
for i=1,#msg do
  helpers.char_array_setitem(zmq_msg,i-1,string.char(string.byte(msg,i)))
end
local res = zmq.zmq_send(sendto, helpers.char_p_to_void_p(zmq_msg), #msg, 0);
zmq.zmq_close(sendto);
zmq.zmq_ctx_destroy(context);
sleep(1)
```

FIGURE 7: Push-pull: the code of a push worker (2 of 2)

```
print("BEGIN PULL"..os.date())
local _core= package.loadlib("./core.so","luaopen_core")
if not(_core) then print("error pthread") os.exit(1) end
local pthread = _core()
local _core= package.loadlib("./zeromq/core.so","luaopen_core")
if not(_core) then print(th .. ":error zmq") return 1 end
local zmq = _core()
local f = package.loadlib("helpers/core.so","luaopen_core")
if not(f) then print("Error on loading helpers" ) return 1 end
local helpers = f()
local clock = os.clock
function sleep(n)  -- seconds
  local t0 = clock()
  while clock() - t0 <= n do end
end
local attr  = pthread.new_pthread_attr_t_p()
local max_workers = 100 --512
local limit_socket = 512 -- FD_SIZELIMIT  ?
if max_workers>limit_socket then max_workers=limit_socket end
local th = {}
for k=1,max_workers do th[k]=pthread.new_pthread_t_p() end
pthread.pthread_attr_init(attr)
pthread.pthread_attr_setdetachstate(attr, pthread.PTHREAD_CREATE_JOINABLE)
local data = nil
local worker_skeleton =
[=[
  local th = "%s"
  local f = package.loadlib("helpers/core.so","luaopen_core")
  if not(f) then print("Error on loading helpers" ) return 1 end
  local helpers = f()
  local _core= package.loadlib("./zeromq/core.so","luaopen_core")
  if not(_core) then print(th .. ":error loading _core") return 1 end
  local zmq = _core()
  local clock = os.clock
  function sleep(n)  -- seconds
   local t0 = clock()
   while clock() - t0 <= n do end
  end
  if swiglib_pthread_data == nil then print("error with swiglib_pthread_data") return 1 end
  -- Socket to send messages to
  local context = zmq.lightuserdata_touserdata_void_p(swiglib_pthread_data)
  local sendto = zmq.zmq_socket(context, zmq.ZMQ_PUSH);
  local rc     = zmq.zmq_connect(sendto, "inproc://workers");
  if (rc~=0) then print(th.." worker:error on connect inproc://workers") return 1 end
  local msg = "123456_"..th
  local bufsize= #msg+1
  local zmq_msg= helpers.new_char_array(bufsize)
  helpers.char_array_setitem(zmq_msg,bufsize-1,"\0")
  for i=1,#msg do
    helpers.char_array_setitem(zmq_msg,i-1,string.char(string.byte(msg,i)))
  end
  local res = zmq.zmq_send(sendto, helpers.char_p_to_void_p(zmq_msg), #msg, 0);
  zmq.zmq_close(sendto);
  sleep(1)
]=]
local worker={}
for k=1,max_workers do worker[k]=string.format(worker_skeleton,tostring(k)) end
```

FIGURE 8: Push-pull with threads (1 of 2)

```
-- Socket to receive  messages on
local context  = zmq.zmq_ctx_new();
local recvfrom = zmq.zmq_socket(context, zmq.ZMQ_PULL);
local rc       = zmq.zmq_bind(recvfrom, "inproc://workers");
if (rc~=0) then print("error on bind inproc://workers") return 1 end
data = context
for k=1,max_workers do
 pthread.swiglib_pthread_create(th[k],attr,-1,worker[k],string.len(worker[k]),data)
end
local c={}
local i=max_workers+3 -- uhu a sign of bad design in  push-pull
while (i>0) do
  local size = -1
  local bufsize = 256
  local _buffer     = helpers.new_char_array(bufsize)
  helpers.char_array_setitem(_buffer,bufsize-1,"\0")
  local buffer = helpers.char_p_to_void_p(_buffer)
  --msg_size doesn't include the trailing zero "\0"
  msg_size=bufsize-1
  size = zmq.zmq_recv(recvfrom, buffer, msg_size,0);
  local data=""
  if size==-1 then
    data=""
  else
    if size >(bufsize)  then
      size=bufsize-1
    end
    buffer = helpers.void_p_to_char_p(buffer)
    helpers.char_array_setitem(buffer,size,"\0")
    local t={}
    for i=0,size-1 do t[#t+1]=helpers.char_array_getitem(buffer,i) end
    data=table.concat(t)
  end
  c[#c+1]=string.format("SINK: seen %s size=%s data=%s,#data=%s",i-3,size,data,#data)
  i=i-1
  if i==3 then sleep(1) ; i=0 end
end
for k=1,#c do print(c[k]) end
zmq.zmq_close(recvfrom);
zmq.zmq_ctx_destroy(context);
print("END PULL"..os.date())
```

FIGURE 9: Push-pull with threads (2 of 2)