Is TEX really slow?

The question

Sometimes you read complaints about the performance of TEX, for instance that a LuaTEX job runs slower than a pdfTEX job. But what is actually a run? Consider the following example (in ConTEXt speak):

```
\starttext
    Hello \TeX\ {\bf world}!
\stoptext
```

In the next few pages I will try to explain what happens when you process some text and why even such a simple TEX job takes about half a second to process on my laptop.

Starting up

When we start up the TEX engine (one of pdfTEX, XƎTEX or LuaTEX), the first thing the program does is figuring out in what environment it is running. How is the TEX resource tree organized and what format file is to be loaded (we assume a production run here)? The format file, which is just a saved memory dump, relates to a macro package and after it has been loaded additional loading can happen, triggered by the macro package. A ConTEXt format file for LuaTEX is 11.6 MB (11.0 MB for LuajitTEX), 8.3 MB for pdfTEX and 4.8 MB for XƎTEX. Just for the record: creating a MkIV format for LuaTEX takes 4.8 seconds (a bit less for LuajitTEX), making a MkII format for pdfTEX needs 10.8 seconds and for XƎTEX uses about 6.9 seconds, just to indicate the spread. If you're a LATEX user, can you predict the relative values for that package?

In LATEX, before you start the text in your document, you load one or more packages, while in ConTEXt everything is already loaded. So, due to the much smaller format file normally LATEX wins here in terms of speed, unless you start adding lots of functionality via packages. Before the actual typesetting begins for sure a couple of fonts have to be loaded and the math subsystem has to be set up. All this takes time but with ssd disks and plenty of memory on a modern machine with proper caching of files, it happens quite fast. Next time you reboot your machine, just compare the first TEX run with a successive one and you will see how fast disks and proper file caching help. When you have a short document, the startup time matters, but when you have a document of 200 pages, it can often be neglected. Nevertheless it helps accepting performance penalties when you consider what happens.

The time spent on initializing the file system depends on the size of the TEX resource tree that you use. If you decided to install all there is, you pay a price because the file databases are large. On the other hand, if you consider leaving out plain TEX or ConTEXt, you're not saving much. For instance, if you use an eight bit engine and therefore eight bit fonts, and when you then also want to typeset some cjk text, you end up with huge (for instance) ttf fonts being turned into a collection of small fonts. We're talking about hundreds of extra files here. And in a TEX tree there can be many such font collections. Another example is the TEXGyre collection. When you use a wide engine (Unicode aware) you can do with a dozen fonts (otf), but when you use an eight bit engine, you end up with hundreds of files (tfm, vf, afm, pfb, map files, etc.), each for a specific encoding. So, if speed matters: try to be lean and mean. When I started with using LuaTEX in ConTEXt, we already were using so called minimals with pdfTEX: an as small as possible TEX tree. By using Lua instead of the built-in libraries we could actually speed up the startup even more.

Loading the format itself is basically just copying bytes to well defined memory locations. The only factor that can delay this is when we use formats that are portable across operating systems and hardware architectures. In that case we get a hit from byte swapping: little endian becomes big endian. The interesting fact here is that most users are on platforms that suffer from this swapping, but the good news is that nowadays distributions don't use portable formats. When we found out that this delayed format creation and loading, in LuaTEX we therefore permanently disabled this feature. We also compress the format (zip level 3) which speeds up loading too. One reason why a pdfTEX format is relatively large is that it is not compressed while it has a lot of hyphenation patterns embedded too. The reason why a ConTEXt format file for LuaTEX is much larger than one for XƎTEX (which also has patterns) is that in MkIV (which is ConTEXt for LuaTEX) we have quite a lot of Lua code and also quite some metadata about for instance characters, something that XƎTEX gets from elsewhere.

So, when the bytes in the format file have become tokens in TEX memory space, the job can start. Loading a bunch of extra macros is pretty fast, and often not measurable, but there are exceptions. If you load for instance `tikz` a truckload of files gets looked up and loaded, and that takes time. Loading additional files is not just copying bytes, but involves converting them to tokens, storing them in memory, maybe doing some calculations, etc. It sounds bad, but you just get what you ask for and the rewards are probably worth it. The only difference between engines in this stage is that XeTEX and LuaTEX are using utf which involves a bit more work in parsing the input.

Nowadays fonts are seldom preloaded so the ones to be used have to be read from disk and converted into a suitable format for TEX. And this is again where differences between engines start showing up. Loading a tfm file is pretty fast as it's just some byte juggling and there are not that many bytes involved: widths, a limited set of heights and depths, possibly a handful of ligature definitions and some kerning pairs. A wide engine has to interpret a rather large OpenType font resource and filter the information that it needs: dimensions but also OpenType features (although this depends on how these are processed). Some of these properties are passed from Lua to TEX. However, when you use more than basic Latin, loading these much larger files pays off because we don't need to deal with specific encoding related instances.

Somewhat related is loading of hyphenation patterns. In LuaTEX this is delayed so you only load what is needed but otherwise these are part of the format file and the more languages a package supports, the more get loaded. Because nowadays patterns are defined in utf this means that making a format file (which doesn't happen often) takes more time in pdfTEX than in the other engines, because this encoding is to be somehow mapped onto a (bunch of) eight bit encoding(s). This brings us to the users input.

Often TEX is used for English documents but what if your document is encoded in utf and has substantial amounts of Greek, Cyrillic or Chinese? In that case you end up with lots of two and three byte codes. In pdfTEX this can be handled by making the first byte an active character (a command) that then looks ahead and interprets the following bytes. This is often not enough because each code range might demand its own font, so switching fonts is needed too. In XeTEX and LuaTEX that comes for free. So, conclusions about LuaTEX being slower than pdfTEX depend on the language and script you use! For instance, in pdfTEX typesetting Arabic is macro magic, while in LuaTEX Lua and in XeTEX the engine do the work.

We already mentioned math. Here it depends on the way math is supported in a macro package. Do we use many families and eight bit fonts, or do we use OpenType math, or do we use a mixture? Do we set up most and store it in the format, or do we delay this till runtime? Do we want to mix different math fonts in one document? Do we use regular and bold (heavy) math? Do we support bidirectional math?

Anyway, by the time we're past `\starttext` and can start with typesetting, we already let the engine do a lot. When you're staring at your console, also realize that this all happens in an interpreted language, where macros get expanded, token lists get created and destroyed and all calculations happen by interpretation. Often TEX looks ahead and has to push back tokens into its input. And, grouping means that we have a save stack so that after a group ends adapted registers have to be restored. Although it is not strictly true, you can consider TEX to run on a virtual machine with a large instruction set. And as the meaning or macros can change any time, there is not that much just-in-time optimization possible. If you say `\tracingall` at the top of you document you get an idea what we're talking about.

Typesetting text

When TEX sees the `Hello \TeX\ {\bf world}!` the `H` tells the engine to start a new paragraph. That itself can trigger a lot because a macro package can hook all kind of actions into `\everypar`. But, when that is done, TEX starts consuming the characters that make up the paragraph and it expands macros on its way till `\par` or an empty line is seen. At that moment characters, kerns, penalties, glue, rules . . . all became nodes that got appended to the current node list. When done with that TEX will launch the par builder.

There is a conceptual difference between pdfTEX and LuaTEX with respect to the paragraph builder. I can't speak of XeTEX as I don't know how that works internally but it can't be far from either of these two. In pdfTEX constructing the so called node list and figuring out line breaks is interleaved with hyphenation, font kerning and ligature building. Traditional TEX is very optimized to do only what is needed here.

In LuaTEX these stages are split: we collect, then we hyphenate, build ligatures, kern glyphs, and then break the result into lines. Each is optional and can be overloaded by callbacks (which is why there are split stages). This is how for instance OpenType font features can be applied and special demands of scripts can be met: by intercepting the node list at certain points in the process. Performance wise, pdfTEX is the winner here. But, only when we're talking basic Latin. Anything more complex, and especially a mix of languages and scripts pays a price. Even then XeTEX and LuaTEX can be slower simply because more advanced font features are applied. Quality carries a burden (although today's documents in most cases don't look much better than before).

The paragraph building itself is more or less the same in the engines: first TEX tries without hyphenation. Keep in mind that in LuaTEX the list always is in a hyphenated form, i.e. has discretionary nodes added. If that fails, a hyphenated pass is applied and when TEX is not satisfied, an emergency pass can happen that will stretch or shrink spacing to the extent that makes all happy.

The pdfTEX engine introduced protrusion (hanging glyphs in the margin) and expansion (stretching glyphs so that excessive spacing become less prominent). It uses additional font instances that get created on the fly. In LuaTEX we support the same but don't do it the same way because there we keep information in the glyph nodes. This is more efficient and also gives nicer code. No matter what method is chosen, enabling these mechanisms hit performance. And it looks like some TEXies really think that all will look better so they enable this feature by default, so they always suffer.

As an intermezzo: enabling something by default can always come at a price. A good example is `synctex`, which will add a 5 to 10% overhead to a run. The same is true for inefficient styles. You can load lots of fonts that you never use and it will add runtime.

After the par builder has done its work it hands over the result to the box builders or page builder. There decisions will be made and additional action can be triggered. For instance, the page builder can decide to launch the output routine which is a hook that can do lots of things, depending on the macro package. One of these can be constructing the page body, adding headers and footers and shipping out the page to the output medium, for instance a pdf file.

Shipping out a page is not that spectacular. The page is just a nested linked list that gets serialized to a stream of pdf codes. But, in the process information is collected about what characters from what fonts are used. If used, hyperlinks can be injected. Location specific information can be flushed to an auxiliary file. The more features you enable the more runtime is involved. There is probably not that much difference between the engines here.

Because in LuaTEX nearly all steps can be replaced or extended by callbacks (Lua functions) a macro package can add its own overhead. And here is where comments about performance become somewhat lame. The more you hook in, the more overhead is added. And the worse the code is, the larger the penalty. One cannot blame an engine for that. Because the LuaTEX engine itself is quite efficient, users (or macro packages) can add seconds or even minutes to a run. It is often easier to blame LuaTEX than your own programming skills and/or demands.

This effect is not limited to callbacks. Macros and rendering feature related mechanisms can be inefficient too. A macro package writer can pay attention to that, but a user can have a dramatic impact by adding bad macros, redundant font switches, useless calculations etc. It can really add up! Of course this also relates to how often something is used. For instance, an image inclusion subsystem can involve lots of (possibly inefficient) code, but because the number of images is normally small it has no significant impact on the run. The final inclusion of the resource will definitely have more impact.

In the small example text above, not much is happening: only a font switch. But again, a simple `\bf` can be either a straightforward switch to a font (basically changing the current font id) or it can involve some more: housekeeping, loading a font, triggering related mechanisms to also adapt to bold, etc. In most cases one can assume that a macro package writer has done a decent job on it.

In this small sentence we see the `\TeX` macro. The original definition by Don Knuth is not that complex but still involves moving glyphs around. In ConTEXt the definition is such that the rendering adapts itself to the current font as good as possible. You really don't want to see the full expansion (10.000 such expansions take half a second so in practice it goes unnoticed.)

Wrapping up

Once we've arrived at the `\stoptext` the engine needs to wrap up the result. At that moment, it is known which fonts are used and what characters from these fonts are referred to. The shapes of the used glyphs need to be embedded. Normally this process is quite efficient, but just as one can see some hiccup before `\starttext`, an extra hiccup can happen after `\stoptext` (or whatever your macro package uses).

After the run, a decision has to be made about successive runs to get the table of contents right, fix cross references, sort registers, massage bibliographies and more. In ConTEXt dealing with this is part of the regular run but one can also delegate this to an external program. Anyhow it adds to the runtime (or time between runs). Macro packages differ in the way they deal with this.

Conclusion

In ConTEXt performance is measured in pages per second. An average document does between 20 and 30 pages per second. Of course when you need multiple runs the effective average drops. But, because input can also be for instance xml, performance is then also influenced by interpreting this format. And, when your pdf needs to be tagged, again some overhead can be added (typically you can delay that overhead till the final run). When you have all kinds of MetaPost code, some runtime gets added (but not much) even when that happens realtime during typesetting. When you use color or backgrounds, in text or tables or in the page body, it comes at a price. But, even then, runtime is still acceptable: processing the 300 page LuaTEX manual on my laptop currently takes some 13 seconds and although processors don't become faster I bet that on a more modern machine it goes below 10 seconds (maybe even lower than I expect) but I can't test that right now.

So let's summarize the above. When you feel that your TEX job runs slow, try to answer these questions:

- ☐ What engine do I use, an eight bit or a Unicode aware one?
- ☐ What kind of fonts do I use, eight bit (Type1) or OpenType?
- ☐ How much do I ask from the font system?
- ☐ Do I really need expansion and/or protrusion?
- ☐ How much math magic do I need?
- ☐ Do I really need to load all these extra packages (modules)?
- ☐ Is my styling okay?
- ☐ Are my own macros, when used in abundance, top notch?
- ☐ Do I really need to enable all these features now?

Quite likely pdfTEX, XƎTEX and LuaTEX will all stay around, so you can choose whatever suits you best. However, the choice might also depend on to what extent the macro package supports all engines. For ConTEXt there is not much choice. All recent development relates to LuaTEX, so you're stuck with that. The good news is that on average a LuaTEX run is faster than one with pdfTEX or XƎTEX. It also offers way more, which is why most users made the switch.

Typesetting the MetaFun manual could easily take many minutes in pdfTEX, but in LuaTEX it takes less than 20 seconds. We sometimes process collections of xml files, for instance math schoolbooks of hundreds of pages. Thousands of small files are combined runtime based on student profiles and the colorful result has more (small) images than pages, typeset alongside the text. Realtime content selection, xml coding error correction, all happens each run. This takes less than a minute for the few runs needed to construct the document compared to close to an hour in the pdfTEX setup (if it's possible at all). Comparing performance is more than clocking a run.

When pondering processing speed, it might help to think of what actually has to happen when your document gets processed. Then you might also consider how fast a 300 page equivalent webpage (with the same amount of tables, images, math, etc) would render and with what memory footprint (apart from assembling that page). Or, how would a word processor do a change in page 5 that affects all following pages. Or how does a desktop publishing system deal with your work in progress in terms of preview, memory management, generating output, etc. Probably TEX and friends, which are rather robust and reliable, won't come out that bad. So we end with asking ourselves:

- ☐ Is the alternative, using another program, really faster?

An the equally valid question:

- ☐ Can an alternative tool support me in a way that I like?

Hans Hagen