Parallel TEXen mit Python

Uwe Ziegenhagen

In diesem Artikel möchte ich kurz vorstellen, wie mit Unterstützung durch ein Python-Skript die in vielen Rechnern mehrfachen vorhandenen CPU-Kerne dazu benutzt werden können, LATEX-Läufe zu parallelisieren, um so Zeit bei der Übersetzung zu sparen.

Einführung

Der durchschnittliche Computer-Nutzer hat heutzutage auf dem Schreibtisch weit mehr Rechenleistung, als der NASA für den Flug zum Mond bereitstand[1]. Mehrere CPU-Kerne oder sogar mehrere CPUs sowie eine vor 20 Jahren unvorstellbar große Menge RAM langweilen sich die meiste Zeit, weil in vielen Fällen der Mensch vor dem Rechner der limitierende Faktor ist. So schnell, wie ein moderner Rechner rechnen kann, kann in den meisten Fällen der Mensch nicht die Aufgaben heranschaffen, die es zu bearbeiten gilt.

Beim Verarbeiten von umfangreichen TEX-Dokumenten ist es jedoch auch heute noch so, dass auch moderne Rechner einige Minuten brauchen, um beispielsweise die hunderten Seiten einer Dissertation mehrfach zu übersetzen.

Ausgangspunkt dieses Artikels war die Frage, ob nicht durch *parallele* LATEX-Läufe ein Geschwindigkeitszuwachs erzielt werden kann. Außerdem – nur unter uns – dient dieser Anwendungsfall als Rechtfertigung vor meiner Frau, einen lauten und großen Rechner anschaffen zu *müssen*.

Als technische Plattform wird Python 3 genutzt, andere Programmiersprachen und selbst `make`[2] halten jedoch ähnliche Funktionalitäten bereit.

Python-Code

In diesem Abschnitt möchte ich kurz auf den Beispiel-Code in Listing 1 eingehen, den ich basierend auf einem stackoverflow-Beispiel[3] für LATEX angepasst habe. Es werden drei Python-Bibliotheken geladen: `multiprocessing`, `time` und `os`. `multiprocessing` kümmert sich um die Parallisierung der Aufgaben, `time` stellt eine Stoppuhr bereit,

[1] `http://www.computerweekly.com/feature/Apollo-11-The-computers-that-put-man-on-the-moon`

[2] `www.gnu.org/software/make/manual/html_node/Parallel.html`

[3] `http://stackoverflow.com/questions/16181121/python-very-simple-multithreading-parallel-url-fetching-without-queue`

um den Leistungszuwachs messen zu können und `os` setzt die Betriebssystembefehle zum Aufruf von `pdflatex` ab.

Die Variable `files` bekommt eine Liste von zehn identischen TEX-Dateien übergeben, die dann parallelisiert übersetzt werden sollen. Die einzelnen Dateien enthalten nur einige übliche Pakete sowie einige Blindtext-Paragraphen, auf eine Darstellung kann daher hier verzichtet werden. `compileFile()` ist die Funktion, die den `pdflatex`-Aufruf für ein übergebenes Dokument (in der Variable `cfile` gespeichert) steuert. Die Funktion versucht, die übergebene Datei im Batch-Mode (also mit möglichst wenig Ausgabe) zu übersetzen, bei auftretenden Fehlern wird eine Exception ausgelöst und der Exception-Text zurückgegeben.

`start = timer()` startet die Stoppuhr, auf die dann bei den einzelnen Läufen referenziert wird. Die folgende Zeile sorgt dafür, dass der `ThreadPool` erstellt wird, dem dann die Aufgabe sowie die Liste der zu übersetzenden Dateien übergeben wird. Einen `ThreadPool` muss man sich als Sammlung von Arbeitsbienen vorstellen, in diesem Fall acht Stück,was der der Zahl der CPU-Kerne in meinem Rechner und damit der Zahl der maximal *gleichzeitig* parallelisierbaren Aufgaben entspricht.

Der Rest des Codes sorgt nur dafür, dass die gesetzte Zeit – beziehungsweise im Fehlerfall die Fehlermeldung – ausgegeben wird.

```
1  from multiprocessing.pool import ThreadPool
2  from time import time as timer
3  import os
4
5  files = ['test-01.tex', 'test-02.tex', 'test-03.tex', 'test-04.tex',
6  'test-05.tex', 'test-06.tex', 'test-07.tex', 'test-08.tex',
7  'test-09.tex', 'test-10.tex']
8
9  def compileFile(cfile):
10   try:
11     result = os.system('pdflatex -interaction=batchmode ' + cfile)
12     return cfile, None
13   except Exception as e:
14     return cfile, e
15
16 start = timer()
17 results = ThreadPool(8).imap_unordered(compileFile, files)
18 for cfile, error in results:
19   if error is None:
20     print("%r compiled in %ss" % (cfile, timer() - start))
21   else:
22     print("Error compiling %r: %s" % (cfile, error))
23     print("Elapsed Time: %s" % (timer() - start,))
24
25 print('Gesamtzeit',timer() - start)
```

Listing 1: Quellcode für den ersten Test

Parallele Spendenbescheinigungen

Während die Unterschiede im oben erwähnten Beispiel zwischen rein sequentieller und paralleler Bearbeitung nur bei wenigen Sekunden (7,9 Sekunden versus 3,6 Sekunden) liegen, bringt das nächste Beispiel deutlichere Unterschiede. In DTK 2/2014 hatte ich gezeigt, wie man mit Hilfe von LaTeX, MySQL und Python Spendenbescheinigungen setzen kann, der Prozess generiert dabei für jede Spendenbescheinigung eine eigene TeX-Datei.

Für das Jahr 2015 hatte ich in meiner Funktion als Schatzmeister des Kölner Dingfabrik e. V. 88 einzelne Dokumente, also eine gute Basis für einen Test. Als Rechner stand ein Dual-Xeon mit insgesamt acht Kernen, 48 GB RAM und installiertem TeX Live 2016 zur Verfügung, alle TeX-Dateien befanden sich in einer RAM-Disk. Sequentiell dauerte die Übersetzung knapp 60 Sekunden, parallisiert weniger als 8 Sekunden. Ein weiterer Test mit diesen Dateien auf einem i7 unter Windows 8 – mit den Dateien auf einer SSD – ergab rund 74 Sekunden für die sequentielle Übersetzung und ungefähr 24 Sekunden für die parallelisierte Übersetzung.

Fazit

Im alltäglichen Gebrauch ist es vermutlich nicht wirklich notwendig, TeX-Dateien parallelisiert zu übersetzen, in den meisten Fällen ist die sequentielle Verarbeitung sogar notwendig, um die Abhängigkeiten zu `makeindex` und `biblatex`/`biber` aufzulösen.

Bei einer Vielzahl ähnlicher Dokumente, die es zu übersetzen gilt, zeigen sich aber signifikante Geschwindigkeitszuwächse. Darüber hinaus macht es natürlich Spaß, die vorhandene Hardware richtig ausnutzen zu können…

Wie immer freue ich mich über Anregungen und Ideen, gern per E-Mail an ziegenhagen@gmail.com.