PAPO 2015:

Platzierung von Fluggästen

im Flugzeug

Frederik Wille, Alexander Timmermann

{3wille,3timmerm}@informatik.uni-hamburg.de

1. Oktober 2015

Inhaltsverzeichnis

1 Einleitung

Die Problemstellung bestand im Wesentlichen aus der Erstellung, Implementation und anschließenden Parallelisierung eines Algorithmus, der Fluggäste auf Flüge bzw. dann auch auf Sitzplätze im gewählten Flugzeug verteilt. Dabei stießen wir auf einige Schwierigkeiten, die im Report erläutert seien.

2 Modellierung

Die Passagiere: Die Passagiere haben mehrere für die Verteilung relevante Attribute. Sie haben alle eine individuelle ID, ein Flugziel, eine Sitzpräferenz (also "window", "aisle"...), eine Sitzklasse (Economy, Business...) und eine zeitliche Präferenz (also "morgens", "mittags"...).

Die Flüge: Die Flüge haben natürlich ebenfalls mehrere relevante Attribute, die die der Passagiere widerspiegeln, also ID, Flugziel, Abflugzeit und, sehr wichtig, die Seatmaps, das heißt eine Wiedergabe der im Flugzeug vorhanden Sitzplätze.

3 Lösungsansatz

Die in →Modellierung angesprochenen Attribute sind eins-zu-eins auch so im Code umgesetzt.

Zur Platzierung favorisierten (und implementierten) wir zunächst einen Algorithmus, welcher auf dem Backtracking-Prinzip fundiert.

Einschub

Backtracking bezeichnet eine Methode, mit der nach dem trial-and-error-Prinzip eine optimale Lösung für ein Problem gefunden werden soll. Dabei soll aus einer Teillösung eine optimale Gesamtlösung abgeleitet werden. Führen die letzten Schritte nicht zu einer optimalen Lösung, werden sie zurückgenommen (daher der Name) und es werden alternative Wege erforscht.

In der Rückschau würden wir diesen Lösungsansatz allerdings nicht mehr wählen, aus Gründen die hoffentlich im Laufe des Reports ersichtlich werden.

Die finale Implementation erfolgte dann in Python, unter Zuhilfename der MPI-Bindings für Python (mpi4py).

4 Parallelisierungsschema

Die Passagiere lassen sich sehr leicht nach ihren Attributen in mehrere Gruppen aufteilen, was die Wahl des Parallelisierungsschema sehr einfach macht. Am sinnvollsten erscheint hier eine Aufteilung nach Flugziel, bzw. dann wieder nach Sitzklasse.

```
def split_passengers_by_destination(passengers):
  passengers_by_destination = {}
  for passenger in passengers:
    if not passenger.destination in passengers_by_destination:
      passengers_by_destination[passenger.destination] = []
```

```
    elif not isinstance(passengers_by_destination[passenger.destination], list):
      passengers_by_destination[passenger.destination] = []
      passengers_by_destination[passenger.destination].append(passenger)
  return passengers_by_destination
```

Daraufhin werden die Passagiere entsprechend der Partitionierung mit MPI auf die Worker verteilt.

```
mpi_seats = mpi.scatter(mpi_seats, root=0)
```

5 Laufzeitmessungen

Laufzeitmessungen erwiesen sich als, gelinde gesagt, erratisch. Da das Backtracking per trial-and-error vorgeht, ist zum Start auch bei Eingabe der selben Daten, nicht ersichtlich, wie viele Kombinationen tatsächlich ausprobiert werden müssen, um zu einem Ergebnis zu kommen. Das macht dann natürlich die resultierenden Werte nicht repräsentativ.
Vereinzelt lässt sich jedoch ein Speedup feststellen. Dazu beispielhaft einige Laufzeiten:

Passagiere	1	3	4	5
1 Core	0.01704	0.30381	25.21396	17.59950
48 Cores	0.11672	0.13384	1.65313	1.54274

Es gestaltet sich als schwierig, aus diesen, nicht-deterministischen Daten einen allgemeinen Trend herauszukristallisieren, es kann jedoch angenommen werden, dass die benötigte Rechenzeit mit der Anzahl der benutzten Prozesse sinkt.

6 Leistungsanalyse

Ein Profiling der sequentiellen Variante erwies sich als sehr leicht zu erstellen. Mit cProfile, dem eingebauten Profiler von Python, lassen sich sehr leicht Profilingdaten erstellen. Diese Daten lassen sich dann mit `pyprof2calltree` in ein Format umwandeln, das sich mit `kcachegrind` visualisieren lässt. In →Anhang I ist der Calltree des sequentiellen Programms aufgeführt. Wie man leicht erkennen kann, wird sehr viel Zeit damit verbracht, die möglichen Kombinationen zu kalkulieren, dort wäre also am meisten Potenzial. Leider sind wir hier aus Zeitgründen nicht mehr zu einer umfassenden Optimierung gekommen.

Das Profiling der parallelen Variante hingegen erwies sich als unmöglich. Keines der für Python vorhandenen Werkzeuge ist mit MPI kompatibel, sodass wir hier leider keine Daten erstellen konnten.

7 Skalierbarkeit

Das Programm lässt sich relativ schlecht skalieren. Durch die Wahl eines naiven Algorithmus, der auf Backtracking basiert, steigt die Anzahl der auszuprobierenden Kombinationen exponentiell mit dem Zuwachs an Input an, womit auch die benötigte Rechenleistung im *worst case* exponentiell ansteigt.

8 Verbesserungsmöglichkeiten

Es wird sehr leicht auffallen, dass es eine ganze Reihe an Möglichkeiten gibt, aus dem Problem eine bessere Lösung herauszuholen. Die wichtigsten Möglichkeiten hierbei sind:

1. **Intelligenterer Algorithmus.** Wenn man nicht mit Backtracking arbeitete, sondern mit einem Algorithmus, der nicht möglicherweise erst dutzende oder gar tausende Kombinationen ausprobieren muss, verringerte sich die Laufzeit natürlich erheblich. Vielleicht könnte man etwa einen Approximationsalgorithmus für das Problem finden. Dazu sind wir aus Zeitgründen leider nicht mehr gekommen.

2. **Kein Python.** Python ist eine (subjektiv gesehen) sehr schöne Sprache mit extensiven Features und einer klaren Syntax, aber ein großes Problem ist die Geschwindigkeit. Durch seine konzeptuellen Lücken als interpretierte Sprache wird selbst ein schlecht geschriebenes C-Programm vermutlich schneller sein als ein durchoptimiertes Python-Programm.

3. **Kein mpi4py.** Der Punkt greift natürlich auch (2) auf. Allerdings ist es mit mpi4py sehr schwierig, genaue Leistungsanalysen seines Programms zu bekommen. Möglicherweise kann man, wenn man ein Programm doch in Python schreibt, dann auf eine Parallelisierung mit nativen Threads o.Ä. zurückgreifen.

9 Fazit

Abschließend gilt zu sagen, dass es, wie man in →Verbesserungsmöglichkeiten deutlich erkennen kann, noch sehr viele Möglichkeiten gegeben hätte, das Projekt weiter auszubauen bzw. zu verbessern. Das alles haben wir aus Zeitgründen jedoch leider nicht mehr geschafft.

Anhang I: Calltree