Multithreaded Requests to Cloud Services for Intelligent Address Standardization

Aleksei Sorokin (asorokin@hawk.iit.edu) and Sou-Cheng T. Choi (schoi32@iit.edu)
Department of Applied Mathematics, Illinois Institute of Technology (IIT)

ILLINOIS INSTITUTE OF TECHNOLOGY

Introduction

Motivation: I was driving to IIT for the first time. I input "3333 Soth Wabash Avenue, Chicago" into Google Maps and I received an exact address to which directions were provided. How did Google Maps know the exact address I was referring to? The address I typed in contained a misspelling of "South" and missed state information and zip code. Despite these human errors (noise), Google Maps still identified the exact location. Here lies the basis of *intelligent address standardization*: mapping a noisy address to a clean address that consists of address components such as street number, street name, and city name.

Background: The Python address-standardization software [1] and study [2] compare the accuracies of different geocoding web, or cloud, services. The program applies noise to a clean address before requesting location identification from a web service. A web service's back-end machine-learning models try to identify a user requested address that often does not contain all address components known to the service. A web service's accuracy is tested by counting how many data fields from the clean address match the service's evaluation of the corresponding noisy address. Testing many addresses on a web service gives a more precise average accuracy value of the service. Multiple addresses are sequentially evaluated in a single call via serial processing.

Contribution: We multithreaded the services request and achieved an average speed up factor of 10 for Geocoder, Data Science Toolkit, and U.S. Address (we did not evaluate Google Maps' accuracy in this study as the service started charging for batch requests in July 2018).

Serial Processing

Evaluating addresses one by one in sequential order

Key Software Conceptual Components:

1. Sample Data File: Clean set of addresses
2. Parser: Validates user arguments: number of addresses, cloud service, and noise intensity
3. Request: Makes RESTful-API request to cloud service
4. Mapper: Maps raw results form service to address components
5. Evaluator: Computes accuracies for all addresses

General Process:

1. The Parser reads in user arguments, loads in the first clean address from the Sample Data File, and introduces noise to the address
2. The Request issues a RESTful-API call to the cloud service asking for identification of the (noisy) address
3. When the cloud service comes back with a JSON response, the Mapper extracts the relevant address components
4. The Evaluator counts matching address components between the mapped response and the clean address
5. The Evaluator updates overall matching accuracy
6. The above steps are repeated for all addresses

Strength: Memory usage is low because the same local variables are being reused for every address

Weakness: The program cannot process the next address until the last request comes back with a response. Sometimes the cloud service takes a long time to respond

Multithreading

Evaluating addresses as parallel tasks (threads)

Main Challenges:

1. Speed up performance by simultaneously sending requests in separate threads
2. Track individual and overall accuracies across multiple responses from different threads

Solution:

1. Use a Python multithreading package such as `threading`
2. Provide a user parameter, *n,* that specifies the number of addresses to process per thread (*T*, number of threads = total number of addresses/*n*)
3. Track overall accuracy by combining thread-level accuracies

Key Software Conceptual Components:

1. Sample Data File: Clean set of addresses
2. Parser: Validates user arguments: number of addresses, cloud service, noise intensity, and processing type
3. Multithreader: Creates threads, each of which invokes:
   a. Request: Makes RESTful-API requests to cloud service
   b. Mapper: Maps raw results form service to address components
   c. Evaluator: Computes accuracies for all addresses in a thread
4. Aggregator: Combines results from all threads

General Process:

1. The Parser reads in user arguments and sends them to the Multithreader
2. The Multithreader selects *n* addresses for each thread. In each thread:
   a. Serial processing is preformed by the Request, Mapper, and Evaluator
   b. While a thread is awaiting the cloud service's response, other threads are spawned
   c. Before a thread terminates, it writes the thread-level accuracies to a list shared between all threads
3. After all threads terminate, the Aggregator computes overall accuracy from thread-level accuracies

Strengths:

1. For many tasks, execution time decreases with multithreading
2. The serial processing option is maintained
3. Substantial amount of code reuse

Weaknesses:

1. Memory usage is high as each thread maintains its own serial processing data: multithreading memory = *O*(serial memory × *T*)
2. Program appears to be "frozen" as results are not output until after all threads terminate

Future Work

1. Automatically determine optimal processing type and, in the case of multithreading, value of *n* by a small trial sample (see Figure 1)
2. Predict and inform users, early in the program, approximate runtime
3. Speed up runtime and reduce memory via microthreading on big data
4. Monitor load balancing across processor cores and rebalance load if necessary

Performance Results

Average Response Time: Dependent on run-time server load. A snapshot:

1. Geocoder: 1.3×10^{-1}
2. Data Science Toolkit: 9.3×10^{-2}
3. U.S. Address: 8.0×10^{-4}

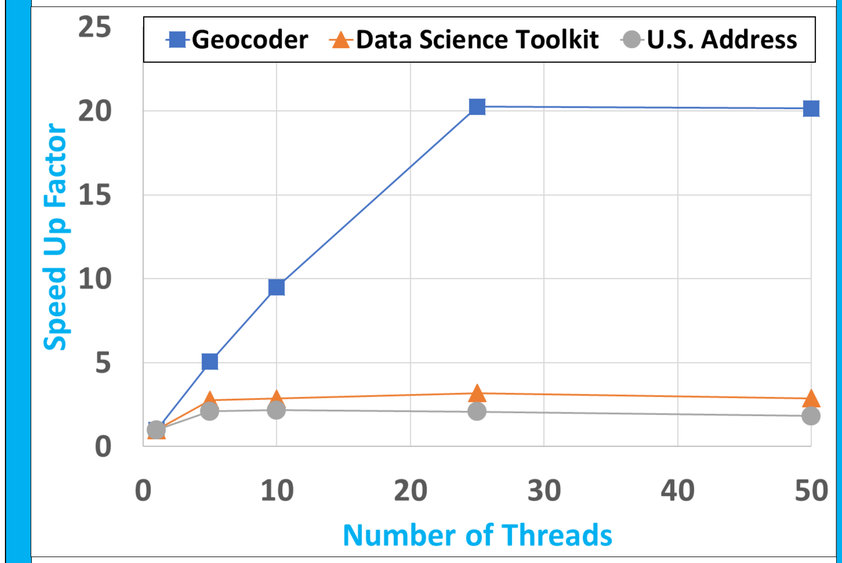

Figure 1: Showing how number of threads affects execution time on a test set of 1,000 addresses.

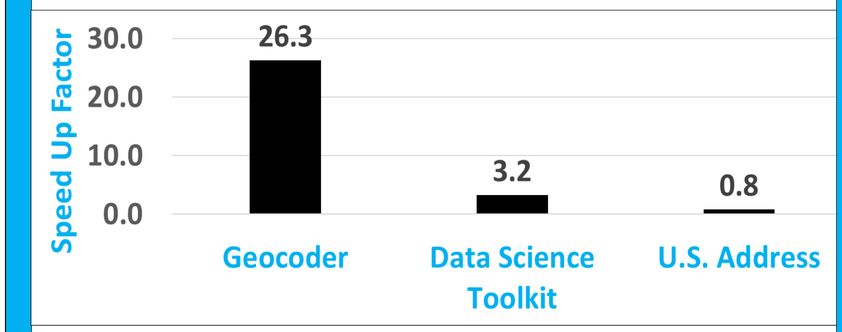

Figure 2: Serial runtime / multithreading runtime on trials of 20,000 addresses, with *n* = 400 (50 threads). Average speed up factor was 10.

References

[1] Choi, S.C.T., Lin, Y.H. (2017). Comparison of Public-Domain Software and Services for Probabilistic Record Linkage and Address Standardization. Python software. Accessed: 2018-08-12. https://github.com/schoi32/prl-splncs

[2] Choi, S.C.T., Lin Y., Mulrow E. (2017). Comparison of Public-Domain Software and Services for Probabilistic Record Linkage and Address Standardization. In *Towards Integrative Machine Learning and Knowledge Extraction* (pp. 51--66), Springer LNAI 10344