Parallelization of the ActivitySim Activity-Based Modeling Framework

Ben Stabler
Jeff Doyle

RSG
the science of insight

ACTIVITYSIM

- An open-source, online, activity-based travel modeling software platform
- Led by a consortium of transportation planning agencies
- New member agencies are welcome to join, and all members help make decisions about development priorities

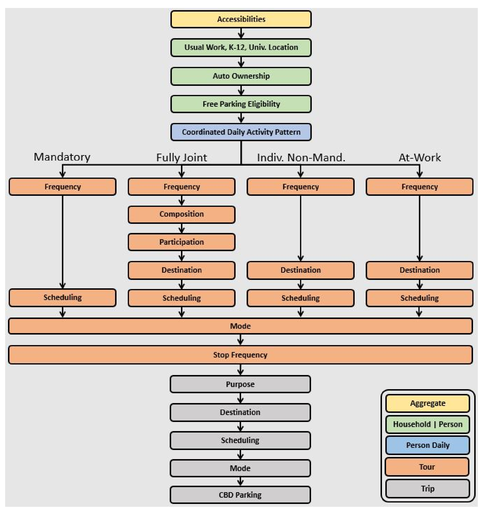
AMPO

activitysim.org

1 ACTIVITYSIM EXAMPLE MODEL

- MTC (San Francisco Bay Area MPO) existing ABM used in production
- Downloadable, runnable, full-scale implementation
- Used for Continuous Integration test system, results verification, and benchmarking
- 7.5 million people, 1450 zones, 825 network skims
- Replica of CT-RAMP model design at this point
- Adding new features every few months

FIGURE 1. Example Model Components



2 VECTORIZATION

- ActivitySim's core design makes use of vectorization
- Vectorization is the processing of an array of data at once instead of each item one at a time
- The core Python pandas and NumPy data science libraries wrap vectorized interfaces around low level C/C++ numerical computing libraries

FIGURE 2. Vectorization Components

COMPONENT	CONTRIBUTION	IN ACTION
activitysim pandas $y_{it} = \beta' x_{it} + \mu_i + \epsilon_{it}$ NumPy python Intel MKL	Models Tables Matrices Glue Math	High-level instructions C/C++ arrays

- Process chunks of rows of household, person, tour, or trip pandas data tables
- Like R, operate on the entire table at once within a single process
- The goal
  - Express all calculations with pandas, NumPy vectorized functions and avoid for loops
  - Write as little raw Python as possible

ARC
METROPOLITAN TRANSPORTATION COMMISSION
SAN FRANCISCO COUNTY TRANSPORTATION AUTHORITY
SANDAG
METROPOLITAN COUNCIL
Puget Sound Regional Council
Oregon Department of Transportation
SEMCOG

3 PARALLELIZATION

- Single-process, the current full-scale example runs in 24 hours and 30 minutes on the test machine
- With 24 threads, the reference model runs in 5 hours on the test machine
- The goal of parallelization is not just speed, but also software that is:
  - Difficult to break
  - May use Python package advances
  - Easy to use and extend

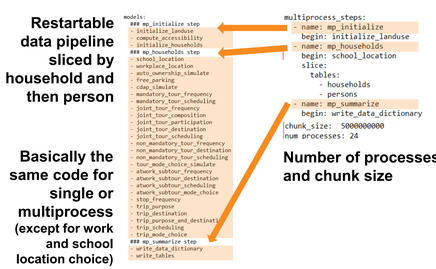
TEST MACHINE

Intel Xeon E5 2.6GHz 28 core 224GB RAM

4 PARALLELIZATION QUESTIONS

- Low-level or high-level? High-level most straightforward since the problem is embarrassingly simple to parallelize
- Process households in parallel since they are largely independent of one another
- Create shared data structures and accumulate results across households when necessary
- By threading within a single process or by using multiple processes?
- Threading in Python is not easy due to the Global Interpreter Lock (GIL), which helps make Python fast and easy-to-use
- The multiprocessing library is the popular alternative, which means parallel Python sub-processes with independent memory spaces

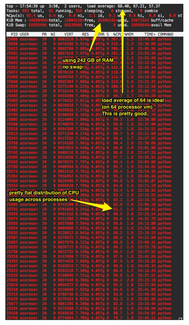
FIGURE 4. Parallelization Design



5 RESULTS

- With 24 processes, the current full-scale example runs in 200 minutes (7.3x faster) on the test machine
- With setting MKL environment variable to override low-level threading, 170 minutes (8.6x faster)
  SET MKL_NUM_THREADS=1
- With MKL=1 and 26 processes, 165 minutes (9x faster)
- Azure Standard_E64s_v3, with 56 processes, 105 minutes, $12
- Azure Standard_M128s, with 120 processes, 84 minutes, $27
- Azure Standard_E64_v3 **Linux**, with 56 processes, 71 minutes, $8
- Azure Standard_M128 **Linux**, with 124 processes, 51 minutes, $16
- Single process 10% sample runs
  - OSX, 15 minutes
  - Linux VM on Mac, 16 minutes
  - Windows VM on Mac, 28 minutes
  - Windows hardware, 28 minutes

FIGURE 5. Results in the Windows Cloud



6 RESULTS DISCUSSION

- Multiprocessing design works well, and surgery was minimally invasive
- Good speed-up, but diminishing returns
- Linux and OSX faster than Windows
- Multiprocessing currently running slower with Python 3, which we'll soon fix
- More research to optimize use of the Python MKL toolkit

7 FURTHER WORK FOR CONSIDERATION

- Existing code improvements
  - e.g., replace costly string operations with faster categorical data operations
- Algorithmic improvements
  - e.g., pre-calculate and cache a fixed set of segmented logsums
- Machine tuning
  - e.g., optimize Windows MKL settings