Adding Multi-threaded Decoding to Moses

Barry Haddow

Fourth MT Marathon, Dublin
27th January 2010

Outline

- Why multi-threaded decoding
- Design of multi-thread moses
- Moses server
- Performance experiments
- Conclusions and further work

The Rise of Multi-core

- Multicore processors are ubiquitous
  - dual-core laptops and desktops are the norm
  - Server grade machines have many more cores available
- Enables several operations to be run in parallel
- Applications should be able to take advantage of the extra cycles
- Parallelism without the admin overhead of grid engine
  - Clusters require more specialist administration, and often don't have enough RAM

The Need for Multithreaded Decoding

- Decoding is a signficant bottleneck in MT experiments
  - Tuning requires repeated decoding

Multi-Process

- e.g. `moses-parallel.pl`
- Extra infrastructure, e.g. SGE
- Copying of models
- Fixed sized chunks

Multi-Thread

- Take advantage of multi-core
- Share models, saving RAM
- Threads can cooperate more closely than process
- Online translation server requires simultaneuous processing

The Need for Multithreaded Decoding

- Decoding is a signficant bottleneck in MT experiments
  - Tuning requires repeated decoding

Multi-Process

- e.g. `moses-parallel.pl`
- Extra infrastructure, e.g. SGE
- Copying of models
- Fixed sized chunks

Multi-Thread

- Take advantage of multi-core
- Share models, saving RAM
- Threads can cooperate more closely than process
- Online translation server requires simultaneous processing

The Need for Multithreaded Decoding

- Decoding is a signficant bottleneck in MT experiments
  - Tuning requires repeated decoding

Multi-Process

- e.g. `moses-parallel.pl`
- Extra infrastructure, e.g. SGE
- Copying of models
- Fixed sized chunks

Multi-Thread

- Take advantage of multi-core
- Share models, saving RAM
- Threads can cooperate more closely than process
- Online translation server requires simultaneous processing

Multithreaded Programming

- *Threads* are separate units of execution within the same process
  - Shared address space
  - Separate stacks
- *Mutexes* or *Locks* are used by threads to synchronise access to shared resources
  - Used to protect shared data structures
  - Thread must acquire mutex before it can enter indicated section of code
  - Other threads are then blocked from entering this section
- Threads can maintain there own copies of a data structure using *thread specific storage*
  - In the boost C++ libraries, this looks like an `auto_ptr`

Multi-threaded moses: Design

- Aimed to minimise changes to existing codebase
- Used threadpool to distribute the work between threads
  - Each thread pulls a sentence from the input, and processes it.
- Main thread-safety issues are:
  - Use of global data structures (`StaticData`), often for convenience
  - Caches - shared read-write data structures - often implemented within layers of indirection
- A mature piece of software such as moses requires a variety of thread-safety solutions

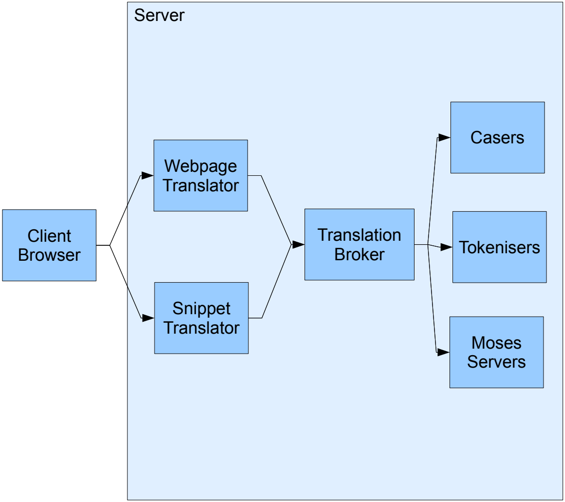
Thread-safety Strategies

1. Remove global data
   - Move sentence-specific data from `StaticData` to sentence-specific `Manager` object
   - No usage of unsafe C-library (e.g. `strtok`)
2. Add appropriate locks
   - Caches for binarised tables, translation options etc.
   - Some amenable to reader-writer locks, but not LRU cache
3. Thread specific storage
   - Used to create per-thread caches.
   - In cases where adding locks would be too disruptive

Moses Server

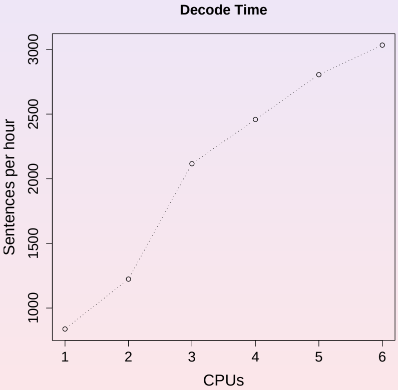
- Server can respond to translation requests over xml-rpc
  - Clients have been created in C++, Java, perl and php
- Uses multi-threaded moses to deal with several requests at once
- Server can also return details on alignments
- Currently used in the statmt demo site `demo.statmt.org`

Case Study - statmt demo



MT Moses Performance - Decoding

- Time taken for europarl model to decode 1023 sentences of news - accounting for startup time.



- Scaling is not linear in number of CPUs
  - Possible resource contention e.g. ram/disk

MT Moses Performance - Mert

- Times (in minutes) for mert, averaged over five runs.

	Plain Moses	MosesMT with 4 threads
Mean iterations	14.6	14.2
Mean total time	1425	689
Mean time per iteration	97.4	46.6
SD time per iterations	16.5	7.1
Mean bleu	33.3	33.4

- Using four threads provides a two-fold (overall) speedup.

Conclusions

- Not hard to extend moses for multi-threaded decoding
- Can make better use of multi-processors
- Easier to use large models
- Speedup is sublinear in processor count
- Disdvantage is less scalable then multi-machine, potential for new types of bugs.

Further Work

- Multi-threaded moses
  - Generation steps
  - randlm/irstlm
  - merge mainlines
  - performance
- Moses server
  - Richer api
  - Configuration switching
  - Architectures for translation systems

Thank you!
Questions?