PBML

The Prague Bulletin of Mathematical Linguistics
NUMBER 93 JANUARY 2010 57–66

Adding Multi-Threaded Decoding to Moses

Barry Haddow

School of Informatics, University of Edinburgh, Informatics Forum, 10 Crichton Street, Edinburgh, Scotland, EH8 9AB

Abstract

The phrase-based translation system Moses has been extended to take advantage of multi-core systems by using multi-threaded decoding. This paper describes how these extensions were implemented and how they can be used, as well as offering some experimental measurements of the potential speed-ups available. Details are also provided of how the multi-threaded Moses library is used to create the Moses server, a platform for building online translation systems.

1. Introduction

A recent trend in computing has been the growth in popularity of *multi-core* processors, able to execute several processes simultaneously. Ordinary desktop and laptop machines are frequently equipped with dual-core processors while servers may have one or more 8-core processors. In order to take advantage of this parallel computing capability, software can be developed to execute with multiple *threads*. Whilst both threads and processes are units of execution, the difference between the two is that threads share the same address space, meaning that multiple threads can access the same in-memory data structures. The consequence is that threads can cooperate more tightly to accomplish a task, but also that the developer must take more care to ensure that common data structures are not damaged by interleaved instructions.

The aim of this paper is to describe some recent modifications to the Moses[1] decoder (Koehn et al., 2007) which enable it to take advantage of this parallel computing capability by decoding several sentences simultaneously in separate threads. Within the typical machine translation (MT) research environment, the main advantage of a

[1] Available under the LGPL from `http://sourceforge.net/projects/mosesdecoder/`

Corresponding author: bhaddow@inf.ed.ac.uk
Cite as: Barry Haddow. Adding Multi-Threaded Decoding to Moses. The Prague Bulletin of Mathematical Linguistics No. 93, 2010, pp. 57–66. ISBN 978-80-904175-4-0. doi: 10.2478/v10108-010-0006-6.

multi-threaded decoder is that it can make more efficient use of the available hardware, enabling quicker decoding. Since the most widespread method for optimising statistical machine translation systems, minimum error rate training (mert) (Och, 2003), involves decoding the tuning set multiple times, improvements in decoding speed lead to faster experimental turnarounds.

The traditional method of parallelising decoding (as implemented in Moses by the `moses-parallel.pl` script) was to split the input file into equal sized segments and send each segment to a separate process, probably running on a separate machine. This method requires a cluster of machines running some kind of job scheduling software (such as Sun grid engine), requiring specialist knowledge to install and administer. It also requires each machine to have access to the translation, language and reordering models, and to have sufficient RAM for the decoder to be able to load them into memory. With the increasing size of the models that are used in MT research, copying these across a network and providing sufficient RAM are non-trivial tasks. The advantage of using threads for parallel decoding is that, since all the parallel execution takes place in the same process, only one copy of each of the models needs to be loaded into memory. Furthermore, it is easier to balance the decoding load between threads than between different processes, as they can cooperate more closely.

Parallel decoding is also essential for the provision of on-line translation services. In this setting, it is clearly undesirable for one user to be blocked whilst another user's translation job is running, and for translating larger blocks of text (such as web pages) it would be useful if some of the sentences could be translated in parallel. Adding multi-threading to the Moses library meant that the decoder could be embedded within a server which is able to process multiple simultaneous requests. Of course, to create a truly scalable online translation system, it is also necessary to allow translation to be spread across multiple machines (Sánchez-Cartagena and Pérez-Ortiz, 2009), as adding more machines an easier way of scaling hardware if the current server's capacity has been reached. Nevertheless, a multi-threaded moses server is an important component in a moses-based online translation system, since it can take advantage of multi-core servers.

The main disadvantage of multi-threaded software is that it can be more complicated to develop, and leads to a new types of bugs which may be difficult to diagnose. In this paper, the techniques used to add thread safety to an existing decoder (namely, Moses) will be discussed, with the aim of providing guidance to others working on similar engineering problems.

The paper is organised as follows: in the following section, techniques for safe multi-threaded programming are described, while Sections 3 and 4 explain the design of multi-threaded Moses and the Moses server, respectively. In Section 6 some experimental results are presented showing the speed-ups possible when decoding with multi-threaded Moses, whilst Section 7 offers some conclusions and suggestions for future developments.

2. Techniques for Multi-threaded Programming

The aim of this section is to briefly introduce some of the concepts and techniques used in multi-threaded programming. It is not meant to be a comprehensive treatment of the topic, merely to provide sufficient background for the design description in the following section.

In most operating systems, programs are executed as *processes* which are separate units of execution (as seen by the scheduler) and have separate address spaces, so they cannot normally access each other's data. A process may, however, have one or more separate *threads*, which are also units of execution with their own call stack, but share the same address space. On a single-processor, single-core machine, threads are mainly used so that the process can continue doing work whilst it is waiting on another task (typically input/output) to complete. However on today's multi-core and multi-processor machines, genuine parallelism is possible.

Allowing multiple threads of execution to access the same memory space is potentially dangerous and can easily lead to memory corruption. To allow safe concurrent access to data structures, a device called a *mutex* (mutual exclusion), or a *lock*, is used to protect critical sections of code so that only one thread is allowed to execute it at any one time. One particularly useful type of mutex is a *reader-writer lock*, which has two modes of locking; one for reading which allows many threads to access the critical section simultaneously, and one for writing where only one thread is allowed to access it.

If mutexes are not used correctly then performance can suffer due to either lock contention or deadlock. The former is where threads hold on to mutexes for longer than is necessary, thus reducing performance because many other threads may be waiting on the mutex to continue performing their tasks. The latter situation can arise, for example, where a first thread acquires lock `A` and then lock `B`, whilst a second thread attempts to acquire the same locks in the opposite order. Depending on the timing of lock acquisition, each thread can be left waiting for the lock that the other one holds, and so neither will be able to continue executing.

As an alternative to the coordinated sharing of data using mutexes, it is sometimes appropriate for each thread to have its own data, for example to provide a thread-specific cache. On many platforms developers implement this using a construct called *thread local storage*. Conceptually, this can be thought of as a common pointer, which when dereferenced returns a block of memory which is unique to the calling thread.

It is possible to create and destroy threads whenever needed, however, creating and destroying threads can be expensive (if they use a lot of thread specific data) and it may be necessary to limit the number of threads active at any time. To avoid constant creation and destruction, threads are often organised into a *thread pool*, whose size may be fixed or subject to upper and lower bounds. A work queue can be used in conjunction with a thread pool to allocate work to the threads, by queueing up the tasks and allocating them to the next available thread.

Programming languages and operating systems differ in the amount of support they offer for multi-threaded programming. In general, Java has very good multi-threaded support, having been created as a multi-thread enabled platform right from the start, and possessing a broad range of thread synchronisation primitives, as well as thread-safe data structures and classes for implementing typical threaded programming patterns. In C++ the multi-threaded support is not as good, with no single standard library for multi-threaded programming, and many platform specific thread libraries. However, there are some mature cross-platform libraries for C++ which include all the appropriate multi-threaded programming primitives, such as ACE[2] (Adaptive Communications Environment) and boost[3]. In addition, there is OpenMP[4], a cross-platform API supported by many leading software vendors which is implemented mainly using compiler directives. In multi-threaded Moses, the boost libraries were used since they offer the required primitives in a cross-platform library which is steadily being incorporated into the C++ standard.

3. Multi-threaded Moses

The aim of this section is to explain the changes that were made to Moses in order for it to support multi-threaded decoding. The threading model adopted for multi-threaded Moses assigns each sentence to a distinct thread so that each thread works on its own decoding task, but shares models with the other threads. This design was chosen to minimise the data sharing between threads.

In making the required changes for multi-threading, one of the considerations was to cause as little disruption to the existing codebase as possible, so the design decisions are not necessarily the same as those that would be employed when building a new piece of software. It was important not to introduce extra dependencies to the Moses build, except where necessary, so the thread-safe version of the Moses library is only built when the appropriate compiler directives are switched on. The work involved in adding multi-threaded decoding to Moses can be divided into two parts; updating the Moses libraries to be thread-safe, and adding the thread creation and management to the Moses mainline.

3.1. Moses Library

To enable the multi-threaded decoding, the Moses libraries need to ensure that, when two different threads are processing their respective sentences, they do not attempt to modify data structures potentially being used by the other threads. The principal shared data structures used in decoding are language models, translation

[2] http://www.cse.wustl.edu/~schmidt/ACE.html

[3] http://www.boost.org

[4] http://openmp.org/wp/

models and reordering models, and at first sight one might think that all the decoder needs to do is read from these data structures, in which case there would be no issue with simultaneous access. However the extensive use of caching within Moses, necessary to reduce levels of disk access during decoding, meant that the data structures representing the models were not necessarily read only. Furthermore, Moses tended to rely on the global singleton object `StaticData` to store data connected with the translation process, even if it was only relevant for one sentence.

The first strategy employed to ensure the thread-safety of the Moses libraries was to use the `Manager` object to store sentence specific data, rather than `StaticData`. An instance of the `Manager` is created for each sentence to be translated, and only contains data relevant to that particular sentence. So in the 'thread per sentence' model employed in multi-threaded Moses, these objects can only be accessed by one thread at a time. The disadvantage of using the `Manager` object to store sentence-specific data is that it must be made available at all points at which this data is needed, thus cluttering up interfaces. In the thread-safe Moses, the `Manager` is now responsible for the pre-loaded portion of the translation table pertaining to its sentence, as well as certain debug data such as timing information.

The translation table (phrase dictionary) in Moses can either be loaded it to memory or utilised in a 'binarised' (on-disk) mode. The former presents no thread-safety issues since it is just loaded into memory at decoder start-up, and is used in a read-only fashion. However, with large translation models it is usual to compile them into a binary format and use them in the on-disk mode, which means that some caching is necessary to reduce the amount of disk access. The system-wide disk cache would be of some help here, but a cache that works at the phrase level is more effective.

The binarised translation model is controlled by the `PhraseDictionaryTree` class which is really just a wrapper for the `PDTImpl` class, the actual implementation. Since the latter is is a read-write data structure, it needed modification to allow concurrent access, and in order to minimise the code changes involved it was decided to use thread specific data to make sure that there was only ever one `PDTImpl` object per thread. The thread specific data class in the boost library has the advantage that it has the same interface as an `auto_ptr`, making it easy to switch between two using compiler directives.

A third solution to the problem of allowing multiple threads to simultaneously access data structures was to use mutexes. For example, the `FactorCollection` object (essentially a vocabulary cache) is now protected by a reader-writer lock so that multiple threads can read from it at any one time, but if a thread wishes to write to it then it must obtain an exclusive lock. For the translation options cache held in the global `StaticData` object, a single mutex is used to synchronise access to the cache. As this is an LRU (least recently used) cache, it must update a timestamp every time an item in the cache is accessed, so a reader-writer lock is not appropriate here.

3.2. Mainline

In order to run multi-threaded decoding, the Moses mainline must create threads and organise the assignment of decoding work to threads. Due to the multiplicity of input/output options in the existing mainline, it was decided that it would be easier to create a new multi-threaded mainline (`MosesMT.cpp`) rather than updating the old one. This has resulted in some undesirable duplication of code, complicating regression testing, which hopefully will be resolved in a future refactoring.

A UML sequence diagram for the important parts of the new Moses mainline is shown in Figure 1. The mainline creates a `ThreadPool` object whose job it is to manage a pool of worker threads. The specified number of threads is created on construction and then jobs are submitted to the pool using the `Submit()` method until the `Stop()` method is called which causes the pool to stop accepting new work, flush the queue and stop all the threads. The unit of work processed by the thread pool is represented by a `Task` object, which in the multi-threaded decoder contains a single sentence to be translated. The tasks are queued up in the pool and as threads become available, they pop a task off the queue and execute it.

When multi-threaded Moses is processing a file, the file is read in, split into lines, and placed in the `ThreadPool`'s queue as a series of `Task` objects. Since these may be executed out of order, it is necessary to put the translated sentences into the appropriate order before outputting them. This reordering is performed by the `OutputCollector` class which uses the input line number to order the sentences correctly.

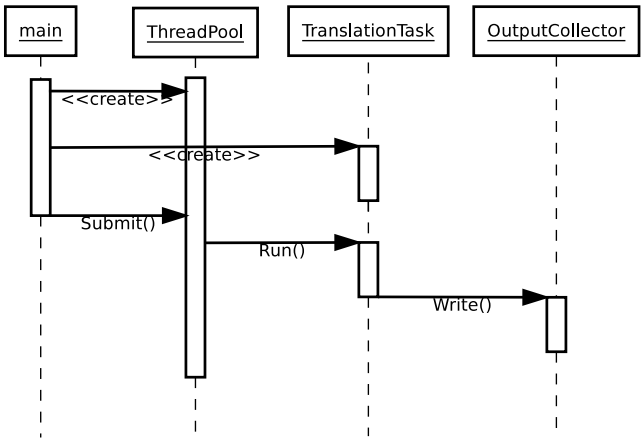

Figure 1. UML Sequence diagram for multi-threaded Moses mainline

4. Moses Server

The main purpose of the Moses server is to enable network access to a Moses-based translation system, for example to build an on-line demo. Making the server multi-threaded offers the advantage that it can process translation requests from more than one user simultaneously, and also it can decode multiple sentences in batches, for example when translating a web page.

The Moses server use the xmlrpc[5] protocol to communicate with its clients. This protocol has the advantage of having mature implementations available in many programming languages; the Moses server has been used with clients written in java, perl, python and php. The specific implementation used in Moses is xmlrpc-c[6].

Since the xmlrpc implementation takes care of managing the server infrastructure, for example listening for client requests and running a thread pool to deal with these requests, the Moses server code only has to implement the remote procedure calls (rpc). Currently the only call that the Moses server implements is the `translate()` call, which receives an input sentence in its `text` field, and returns the translated text in the same field. If the `align` flag is switched on in the method call then the phrase alignment is returned as a sequence of `(target-start, source-start, source-end)` index triples, in target order.

5. Usage

Using multi-threaded Moses is straightforward. The new mainline (`mosesmt`) is intended as a drop-in replacement for the existing mainline. It responds to exactly the same arguments as `moses` and adds a `-threads n` argument to specify the number of threads. Increasing the verbosity of multi-threaded Moses is not recommended as some of the debug code uses non-threadsafe global variables, and the debug messages will be interleaved and difficult to read anyway.

The Moses server mainline (`mosesserver`) also accepts all the usual Moses arguments and adds two of its own. The argument `--server-port n` is used to specify the port on which the server listens, and the `--server-log` can be used to specify a log file for the server to write to. For extra diagnostic information, set the `XMLRPC_TRACE_XML` environment variable before launching the server.

6. Experiments

In this section the results of some timing experiments are presented, comparing multi-threaded and single-threaded Moses. The experiments were run on a Dell PowerEdge server with 4 Intel Xeon Quad Core processors (so it has 16 cores), and 32GB RAM.

[5] `http://www.xmlrpc.com/`

[6] `http://xmlrpc-c.sourceforge.net/`

For the first set of experiments, decoding speed of single and multi-threaded Moses was directly compared, using a translation model similar to the Edinburgh French-English submission for the WMT2009 shared task (Callison-Burch et al., 2009; Koehn and Haddow, 2009). This includes translation models and reordering models trained on all the shared task parallel data, plus a language model trained on the English side of this data, interpolated with the monolingual news data. The translation and reordering models were used in binarised (on-disk) format, but the language model was loaded into memory.

The experiment consisted of decoding the news test set from this shared task (3027 sentences) using plain (single-threaded) Moses, and using multi-threaded Moses whilst varying the number of threads from 2 to 6. In order to account for the fixed costs of loading the models into memory and initialising other data structures, a decoding run was also done for one sentence. Decoding was repeated five times for each type of decoder. The mean times (in seconds) are shown in Table 1.

Decoder	Full corpus	One sentence	Difference	sd(Difference)
Plain Moses	4677	282	4395	623
Moses MT 2 threads	3292	283	3009	505
Moses MT 3 threads	2024	284	1740	154
Moses MT 4 threads	1781	283	1498	100
Moses MT 5 threads	1591	278	1313	37
Moses MT 6 threads	1492	278	1214	45

Table 1. Decoding time (in seconds) for single and multi-threaded decoders, averaged over five runs. The second last column is the difference between the first two, in other words the time to decode 3026 sentences not including start-up and shut-down times. The final column shows the standard deviation of this 3026 sentence time.

From Table 1 it can be seen that there is around a speed increase of around 3.5 going from single-threaded Moses to multi-threaded Moses with 6 threads. Whilst this speed-up is clearly useful, the question arises of why there isn't a six-fold increase in speed. The most likely answer to this question is some sort of resource contention; in other words the six threads are not spending all their time decoding but spending some time waiting for other threads to release a resource. One possible type of resource contention is lock contention, where threads spend time in a block state waiting for other threads to release locks, however the only locks used during the decoding are those on the translation options cache, and running experiments with this cache removed results in similar timing behaviour. It is also possible, depending on the hardware architecture, that there is resource contention at the RAM or disk level,

since decoding requires a substantial amount of data to be accessed from the different models employed.

The next timing experiment compares minimum error rate training (mert) runs using single-threaded and multi-threaded Moses. This experiment uses the French-English europarl corpus (Callison-Burch et al., 2009) for training the translation model and 5-gram language model, with the 2000 sentence dev2006 corpus for tuning, and the test2007 and test2008 corpora for testing. The tuning runs were done on the same machine as the first set of experiments, although because of the length of these experiments it was not possible to ensure that the machine remained unloaded throughout this time. Table 2 shows the timings for single-threaded Moses, and Table 3 shows the corresponding timings for multi-threaded Moses, demonstrating around a two-fold speed-up in mert when using 4 threads.

Run	Iterations	Time	Time per Iteration	Bleu
1	17	2054	120.8	33.4
2	12	1258	104.8	33.3
3	14	1362	97.3	33.3
4	14	1172	83.7	33.3
5	16	1283	80.2	33.3
mean	14.6	1425	97.4	33.3

Table 2. MERT times for single-threaded Moses, in minutes

Run	Iterations	Time	Time per Iteration	Bleu
1	15	735	49.0	33.3
2	23	1320	57.4	33.3
3	8	319	39.9	33.5
4	15	615	47.7	33.4
5	10	456	45.6	33.4
mean	14.2	689	46.6	33.4

Table 3. MERT times for multi-threaded Moses (4 threads), in minutes

7. Conclusions and Future Work

This article has described extensions to the Moses decoder which permit multi-threaded decoding, and also allow Moses to be used as a server to run an online

translation system. Experimental results demonstrated that using 6 threads can speed up decoding by 3.5 times, and a two-fold speed-up in mert was demonstrated, when using a multi-threaded decoder with 4 threads. Further investigation is required to determine why the speed of decoding is not linear in the number of threads. The other outstanding task in multi-threaded Moses is to make the generation tables (used in some factored models) thread-safe; these can be addressed using the same techniques as the translation tables.

The Moses server is already being used successfully in the University of Edinburgh's demo site[7]. A limitation of the current server is that a separate server is required for each language pair so, for instance, to deploy both French-English and German-English systems, each server must load its own copy of the English language model. A proposed update to the Moses server would be to allow *configuration switching*, where one server would be able to run more than one translation system, with the choice of translation system to translate a given sentence would be selected by an rpc argument. This arrangement would save on the RAM used to run multiple Moses servers on the same host with the same target language.

Bibliography

Callison-Burch, Chris, Philipp Koehn, Christoph Monz, and Josh Schroeder, editors. *Proceedings of the Workshop on Statistical Machine Translation*, 2009.

Koehn, Philipp and Barry Haddow. Edinburgh's submission to all tracks of the WMT 2009 shared task with reordering and speed improvements to Moses. In *Proceedings of the Workshop on Statistical Machine Translation*, pages 160–164, 2009.

Koehn, P., H. Hoang, A. Birch Mayne, C. Callison-Burch, M. Federico, N. Bertoldi, B. Cowan, W. Shen, C. Moran, R. Zens, C. Dyer, O. Bojar, A. Constantin, and E. Herbst. Moses: Open source toolkit for statistical machine translation. In *Proceedings of ACL Demonstration Session*, pages 177–180, 2007.

Och, Franz J. Minimum error rate training in statistical machine translation. In *Proceedings of ACL*, 2003.

Sánchez-Cartagena, Víctor M. and Juan Antonio Pérez-Ortiz. An open-source highly scalable web service architecture for the Apertium machine translation engine. In *Proceedings of the First International Workshop on Free/Open-Source Rule-Based Machine Translation*, pages 51–58, 2009.

[7] `http://demo.statmt.org`