Pulling Out All the Stops - Part II

By Phil Brooks

Presented at the 19th annual Tcl/Tk conference, Chicago Illinois November 2012

Mentor Graphics Corporation
8005 Boeckman Road
Wilsonville, Oregon
97070
phil_brooks@mentor.com

Abstract--- At the 12th annual Tcl/Tk conference in Portland, Oregon, I presented a paper entitled 'Pulling Out All the Stops' - which concerned using Tcl as a user interface custom calculation engine at the heart of a high performance electronic design analysis package. This talk discussed the efficiency concerns and implementation details that were considered during implementation of this package. One simplifying constraint applied to the design was that, though this is a multi threaded application, a single Tcl interpreter was used and individual threads would access that Tcl interpreter through a mutex lock. seven years later, customers are running more complex calculation codes on systems with more processors. Locking on the single Tcl interpreter now restricts scaling. This paper briefly reviews the original design and then discusses the conversion to a fully threaded design in which one Tcl interpreter per thread allows completely parallel execution of the Tcl calculator.

1 Introduction - A review of the 2005 paper

The 2005 paper discusses implementation of Calibre LVS's Device TVF feature, in which a highly efficient, though quite limited, calculation engine is given the ability to make calls to a Tcl program allowing for more sophisticated programming capabilities. Some examples of the calculation language to Tcl calling mechanism is described. Finally, a set of implementation details that wring maximum performance out of the Tcl interpreter in this situation are described. Specifically, the major performance related recommendations were:

- Make use of TCL_EVAL_GLOBAL.
- The user's Tcl programs are pre-compiled before calculations are started using Tcl_EvalObjv.
- Runtime data access Tcl_Obj objects for arguments passed into function and return values are set up once up front and are reused during each individual call to the device calculator.
- Use Tcl object commands to hand off performance critical processing from Tcl to C++.
- End users are strongly encouraged to write compact efficient Tcl code.

The end of the 2005 paper includes a brief mention comparing Tcl 8.4 MT vs. Tcl 8.3 non MT builds and shows a favorable scaling improvement especially when 4 or more processors are used. The performance scaling that was visible in toy test cases, where the multi-threaded C++ processing was made as simple as possible and the Tcl processing was made artificially complex, bore no relationship to the real world testing on real customer data where the MT C++ part took vastly more time and the Tcl - even locked and single threaded - took hardly any time at all. So, the originally shipped implementation didn't use MT Tcl interpreters due to time constraints, tcl version constraints (we originally released on 8.3), some lurking bugs in the Tcl MT package, and one critical Tcl threading implementation detail that we had yet to uncover!

2 What has happened in the mean time

Over the next several years, customers started using the interface, at first in exactly the way it was designed... The original interface was conceived as a simple functional interface in which customers would pass in a couple of vector style parameters, run through a loop, perform the calculations of their choosing, and return a single value to the calculation engine to be used in later processing. We found that after they got used to the interface, customers did far more clever things than that with it:

- They would pass in dozens of parameters and make complex multi-step calculations.

- They would calculate several different parameters using the same set of input parameters, often doing preliminary calculations repeatedly due to the fact that our interface would only return a single number at a time.
- To get around the single return number constraint, customers would put together string results that were actually a concatenation of numbers separated by '_' characters and then pass them back and reparse those strings in subsequent calls.

We also found, during performance analysis on 16, 32, and 64 way systems, that, especially when customers did these sorts of complex calculations, the single shared Tcl interpreter was having a throttling effect on overall MT scaling of the application.

3 Searching for Clues

Most of the documentation that I could find for using multi-threaded Tcl talked about using Tcl itself to start and manage the threads. There is one passage in the book "Practical Programming in Tcl and Tk" by Welch, Jones and Hobbs [Welch] that says:

> *"At the C programming level, Tcl's threading model requires that a Tcl interpreter be managed by only one thread." p. 322*

And that's about it. The rest of the chapter talks about all of the facilities that Tcl has at the scripting level for creating and maintaining multiple interpreters using the Thread package, how to make them communicate with one another, how to pass channels back and forth, how best to write to a common file, how to share data efficiently between them, synchronization between Tcl threads, but nothing else that talks about how to run multiple interpreters that don't have to interact with one another from C. I didn't know it at the time, but that simple statement contains the most important thing (and possibly the only important thing) that you need to know to create independent parallel Tcl interpreters running on separate threads in C.

4 More about Calibre threading

Calibre has been a multi-threaded application for a very long time. There are embarrassingly parallel problems of computational geometry to be solved, and so they have been solved and improved upon since at least April of 1998. Here is an interesting CVS checkin log:

```
revision 1.1
date: 1998/04/10 17:41:31;
author: ****; state: Exp;
routines related to flat drc thread are going be in this
file. Currently there is not much in it.
```

Needless to say, there is considerably more in that file today than there was and the threading capabilities in Calibre are very well entrenched, so changing the threading model and control of the entire application for one new feature in the LVS is not in the cards. So years after the initial paper on 'Pulling out all the stops', it appeared, in fact, that more stops needed pulling, I rolled up my sleeves and resurrected the notion of running the Tcl interpreters in threads.

5 Task queue and thread pool Pattern

Let's look briefly at this commonly used pattern for threading architecture. The task queue and thread pool pattern [Wikipedia] is also known as the Worker-Crew Model [Sharapov]. In this pattern essentially separates the management of processing threads in a system from the tasks that need to be performed. This means that the threads can be managed for one set of constraints (like hardware availability, licenses, performance management, etc.) while task creation is under the purview of the application algorithms (like what things do I need to do and in what order do they need to occur).

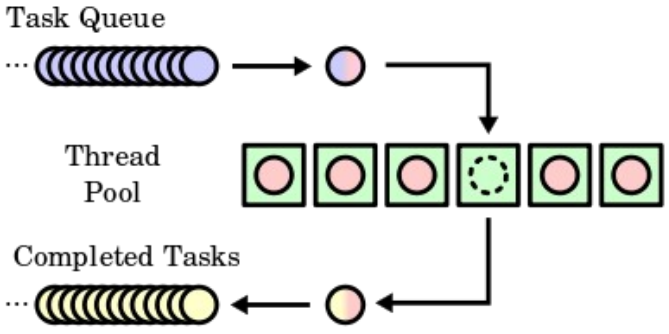

Figure 1. - Task Queue and Thread pool pattern - Wikipedia

Calibre uses just such a model for threading. This allows many disparate algorithms that may execute to solve a customer's particular problem to parcel their work out to a number of worker threads that complete the tasks according to hardware and licensing constraints that are managed by a completely different part of the system.

5.1 An Example Work Queue and Thread Pool API

So a work queue API generally has the following sorts of requirements and capabilities:

- it allows you to package up a unit of work including data and some direction for what needs doing.
- it allows you to ship that unit of work off to the work queue.

The thread pools then grab the pieces of work and execute them. At that point, the essential get-something-done part of the API comes in:

- it allows you to write some code that will do the work in a worker thread.

finally, once the work is done, the final piece:

- wake me up when its over.

5.2 Implementing work queue and thread pool in Calibre LVS

So, the Calibre device recognition algorithm follows the above prescription:

- In the main thread: Package up the work. Ship the tasks off to the thread pool.
- In a worker thread: Wake up and perform a task on a particular set of data.
- In the main thread: OK - wake up, everything is done now.

In my initial implementation, the following steps are taken in the main thread:

- Start setting up the algorithmic tasks. Ship the tasks off to the thread pool.

In a worker thread:

- Use thread specific storage to look for an existing interpreter
- if there is no interpreter create one. Also create the entire infrastructure for the interpreter, feed it the Tcl program text etc. Set up the argument objects, return objects, calling infrastructure etc. Store thread specific data inluding the pointer to the interpreter.
- Execute the local work task - it calls the thread's Tcl interpreter when needed.

At this point, everything is doing just fine. Now, once we finish all of the work, we need to clean up after ourselves.

- Wake up everything is done.
- Clean up the Tcl interpreters.

No big problem, just loop remember where all of the tcl interpreters are and delete them one at a time from the main thread!

5.3 Oops...

For the most part, the initial implementation worked fine. However, as this excerpt from my post to comp.lang.tcl points out, the cleanup at the end wasn't going so well:

> *At the start of the processing that was requiring multiple Tcl interpreters, I set up some thread specific storage in a vector of pointers, then as each thread got its first task, it would check for an interpreter there if there was no interpreter, it would create one under its thread index.*
>
> *Threads would individually calculate, using Tcl and they ran nicely in parallel. At the end of the processing, once all of the tasks were done and the threads all quietly parked in their Calibre threading model parking slots, I would loop through on the main thread and destroy each interpreter. This is where things would go terribly wrong. Crashes, memory leaks, unclosed files, etc..*

I finally reproduced the entire problem outside of my application in a small snippet of C++ and Tcl C api. While doing that, I found out that if I deleted the interpreter from inside the same thread that it was created in, all was fine, but if I did the same thing from the main thread in the same way that I was in the application, I got the same crashes that I had in the full application. I posted my findings to Comp.lang.tcl and received the following reply:

> *Gerald W. Lester: ...*

> *I also found that by deleting the interpreter inside the same*
> *execution thread that it was created in,*
Where else would you be deleting it from -- an interpreter is not supposed to be accessed by more than one thread. You may have many interpreters per thread (i.e. a thread can access/use many interpreters), but only one thread per interpreter (i.e. only a single thread should be accessing a given interpreter).

Let's reread the bit from the book:

> *"At the C programming level, Tcl's threading model requires that a Tcl interpreter be managed by only one thread." p. 322*

Ahh - so that's what it means! I suggest rewriting that sentence:

> *"At the C programming level, Tcl's threading model requires that a Tcl interpreter be created, used and destroyed by only one thread. Interpreters cannot be used across multiple threads."*

6 Conclusion - Tcl threading from C Threads

Using Tcl interpreters in separate threads requires, absolutely, that the threaded interpreters be started (Tcl_CreateInterp) and stopped (Tcl_DeleteInterp) from within the thread that they execute in. This all makes perfect sense if Tcl is managing all of the threads in the application and lower level bits of code go off and do various low level things in a thread safe manner. It is a disaster if you are trying to plug Tcl into an existing threading system that uses the thread pool pattern in which you don't have control of or access to the individual threads, but instead submit tasks to a work queue that eventually dispatches them to threads.

Construction of the interpreters following this rule is fairly easy. Simple lazy creation of the interpreter from inside the execution thread and access through a thread specific variable or array slot works just fine.

Cleanup is much more problematic. Luckily, the guy that writes the work queue and thread pool code for Calibre LVS is just four doors down from me. Also, the architecture of our application is such that there are natural sequence points after each high level operation during which all of the threads are quiet. In Calibre, a new API that implements a "perform this task on each thread" routine which I now use to clean up interpreters. If we had a more heterogeneous work queue, though, this approach might not be possible. It would be much cleaner for this type of usage if

Tcl_DeleteInterp were able to clean up a quiet interpreter regardless of which thread it was created on.

Having found a solution to the Tcl threading issues, we can examine the performance of multiple Tcl interpreters doing user defined calculations in parallel on large numbers of parallel processors where we look for the next performance bottlenecks.

7 Acknowledgements

Special thanks to Fedor Pikus for helping me understand the pthread library more fully. Thanks also to Gerald W. Lester for his response to my initial query on comp.lang.tcl

8 References

"Practical Programming in Tcl and Tk" - Brent B. Welch, Ken Jones, with Jeffrey Hobbs, Prentice Hall, 2003
"Techniques for Optimizing Applications – High Performance" - Garg & Sharapov, Prentice Hall 2002 p. 394
"Thread Pool Pattern" - Wikipedia article