Threading Perl with TBB

Sam Vilain
sam.vilain@openparallel.com
Open Parallel
http://openparallel.com/

May 10, 2011

Abstract

Perl's history has seen two threading models; one shared-state model now discontinued, and one heap-duplicating model with a very inefficient shared state approach. There is also an Erlang-style threading library on CPAN, `threads::lite`, more efficient and scalable but specific in functionality.

A task-oriented parallelisation approach permits parallel operations on data sets as well as pipeline-based programming. `threads::tbb`, the core invention of the paper, uses Intel's Threading Building Blocks (TBB) along with a system of lazy cloning for state, and is shown to result in speed-ups for embarassingly parallel tasks to 8 processor cores or more.

The results are applicable in principle to other languages which are built around boxed variables and state machine interpreters, such as PHP/Zend or standard Python.

The cloning logic duplicates core Perl 5 code in intent, the API for which could be cleaned up to avoid some minor API intrusions. Green fields interpreter approaches would benefit from a `const` concept to avoid duplication in the first place, and for safer operation.

Transformations for task-orientation are similar to those required for event-oriented programming, with potential to parallelize event frameworks, or for APIs which span the two styles.

Contents

1 Threading History **2**
1.1 `5005threads` 2
1.2 Interpreter Threads (`ithreads`) 2
1.3 Deprecation and Retirement of `5005threads` 3
1.4 `threads::lite`, a contemporary approach 3

2 Task Parallelisation and TBB Concepts **4**
2.1 Tasks vs Threads 4
2.2 Parallel Data APIs 5
2.3 Pipeline Tasks 5
2.4 Thread-safe Containers 6

3 API for Perl 5 **6**
3.1 Initializing TBB 6
3.2 Parallel APIs 7
3.2.1 parallel_for 7
3.3 Concurrent container API 8
3.3.1 Overview 8
3.3.2 Lazy deep copying 8

4 Conceptual Findings **9**
4.1 Lazy Deep Copying 9
4.2 Perl 5 CLONE API for XS module writers 10
4.3 Task based vs Event-based programming 10

5 Quantitative Findings **11**
5.1 parallel_for scalability 11
5.1.1 a task not particularly well suited 11
5.1.2 A synthetic task 12
5.1.3 A practical application 14

6 Closing notes **15**

References **15**

1 Threading History

1.1 `5005threads`

The first threading attempt was introduced in unstable Perl 5.004_50 in September 1997 [1]. Recorded in the source history notes as an integration of earlier work, the full details of this are long lost in the sands of time.

The basic idea was to share all interpreter state by default. It tried to make minimal use of thread-private space, using a per-thread data structure which various critical interpreter globals were moved into. This included pointers to the logical stacks (all 5 of them), currently executing instruction, and various per-thread state variables. Much work was also required for threading the regular expression engine (itself another virtual machine of sorts).

By July 1998, the work was considered good enough for a production release and were part of the major new feature bill for 5.005. This threading approach is retrospectively named `5005threads`.

1.2 Interpreter Threads (`ithreads`)

Along the way, people were wanting to use their Perl programs written for unix, which tend to use *fork(2)* a lot, on Windows. Windows does not implement a *fork()* - only threading and starting new programs.

To run Perl scripts written for *fork()*-style multiprocessing, ‘interpreter threads’ were born, starting in November 1999 in Perl 5.005_63 and released in March 2000 with Perl 5.6.0 as an internal/embedding API. The `threads.pm` API evolved against this API and was distributed on CPAN, included with the core in Perl 5.8.0 (July 2002).

With interpreter threads, it was decided to be better to make everything thread-private due to problems encountered with insufficient locking with operations with side-effects. So, interpreter threads use a policy of private state by default, and includes an API call, `Perl_clone()`, for recursive cloning of an entire interpreter's data and execution state.

This API is very expensive in terms of memory; it essentially duplicates the entire heap of the program when a new thread is created, which can easily run into megabytes. Even on a modern system, starting a thread with a few modules loaded can take 10's of milliseconds and obviously increases the anonymous memory size of the process markedly. For emulating *fork(2)* on windows, that is fine.

1.3 Deprecation and Retirement of `5005threads`

Since its inception, bugs were found regularly in the shared state threading engine - usually due to a Perl operation affecting a global as a side effect, and this side effect affecting another thread unexpectedly. The Perl Pumpking, Jarkko Hietaniemi, apparently seemed to give up in desperation, and so declared that the shared state approach was "considered fundamentally broken" in March 2002 (around perl 5.7.3). There were at this point two test failures in the official test suite, though a general impression that "`5005threads` broke CPAN" remained common knowledge [2].

The last version of Perl 5 to build with shared-state threading was 5.8.7 (May 2005), it being inadvertently broken on the `maint-5.8` series by 5.8.8 (February 2006), though apparently nobody noticed. During the 5.9.x development series, the `-Duse5005threads` compile option was removed. So, Perl 5.10.0 (December 2007) is the first production release which officially removed the feature.

1.4 `threads::lite`, a contemporary approach

> This feature is still in evolution. It is eventually meant to be used to selectively clone a subroutine and data reachable from that subroutine in a separate interpreter and run the cloned subroutine in a separate thread.
>
> ---
>
> Perl Pumpking Gursamy Sarathy, 9 December 1999, talking about the possible direction of the Perl `ithreads` API

For a long time, `threads.pm` along with `threads::shared` remained the state of the art for threading in Perl. However, there are serious design issues with `threads::shared` and only the most embarrassing of embarrassingly parallel tasks can be successfully scaled to multi-core systems using it. This is largely down to its shared state implementation.

Shared state via `threads::shared` is implemented via a complicated system of interwoven "magic" data structures, and a dummy interpreter which "owns" the underlying data structures. Data is duplicated out of this tapestry of magic data structures as it is accessed using Perl's magic hooks. Any access - including read access - to the shared data is locked by a single mutex. `threads::shared` could be Perl's answer to Python's Global Interpreter Lock.

Other languages with pure, side-effect free characteristics came to dominate concurrent programming headlines in between. Perl is well known for stealing the best features from other languages, and so inevitably some tried to bring these concurrency approaches to Perl.

`threads::lite` [3] is a notable implementation, and uses the Perl embedding API, along with `Perl_clone()` to start a pool of (optionally) automatically restarting threads, which communicate via thread-safe queues known as "channels". These are similar to key primitives available in Erlang [4]; so long as all state is passed via messages in channels, a high level of concurrency can be obtained.

Unlike `threads.pm`, it does not use `Perl_clone()` on the master process, instead starting up a series of its own worker interpreter objects as requested.

The newly created interpreter is then duplicated via `Perl_clone()` to the level of concurrency requested.

The queues used by `threads::lite` employ `Storable::freeze` and `Storable::thaw` on all but the most basic of messages to marshall data structures into serialized messages. This "clean-room" marshalling method ensures that side effects do not propagate between threads. The work to be done in the worker threads is specified by closures, which are serialised via the `B::Deparse` module and compiled in the worker threads on their start-up.

As this approach has no need for `threads::shared`, one source of contention while passing data around is removed. Further contention caused by allocating memory is likely to start affecting scalability beyond 4 cores or more [5].

2 Task Parallelisation and TBB Concepts

2.1 Tasks vs Threads

The highest throughput approaches to parallelism optimize for keeping all available processors busy doing useful work, and avoiding contention. The explanation for this is a logical deduction: as the processor resource is fixed over a given unit of time, then for a given fixed quantity of workload if all processor cores are constantly working and never waiting then the task will be finished as quickly as possible.

Thread-oriented approaches such as `threads` and `threads::lite` hope to achieve this by starting lots of threads which can carry out work; letting the operating system scheduler manage the workload. So long as there are more threads running than there are available processors, there is usually a high utilisation of available resources.

However, if too many threads are started, per-thread overheads such as stacks or task switching overhead may overwhelm the advantages gained by parallelism. In the context of an interpreted language, the memory overhead for a thread may be significant. Achieving a good balance can be challenging.

Task-oriented Parallelisation such as provided by TBB is an approach that aims to parcel runnable work into discrete chunks which can then be allocated to threads independently. The approach minimizes the number of stacks to the number of threads started globally per-process, typically fixed to the available hardware parallelism. It also allows *task stealing*, which shifts the onus of selecting work to be done onto the task which is able to run; much like members of a smoothly running team will find work to do and just get on with it.

To ease the parceling of runnable work into tasks, several APIs are provided by TBB to cater for a range of parallelisable problems.

It is still possible for overheads to swap useful work. All of these APIs are noted as being most effective when the amount of time spent in an individual task is about 10,000 CPU cycles. Corresponding characteristic figures for the Perl implementation are arrived at in the results section of this paper.

2.2 Parallel Data APIs

Parallel Data APIs are for situations where the work to be performed is on a divisible data set, typically *embarassingly parallel* problems. Embarassingly parallel problems are those for which there is no inter-dependence between processing subsets of the problem.

Examples of parallelisable problems for data sets include iteration (`parallel_for`), iteration with aggregation or reduction (`parallel_reduce`), sorting (`parallel_sort`) and prefix scans (`parallel_scan`) [7].

In TBB, for each of these APIs, first a *body function* is defined which works on a *grain* of data - the *grain size* being either pre-determined or scaled at run-time. The data set must also be indexed by a *divisible range* - which can be an integer range (as in iterating over an array of data), or a divisible set, such as the set of keys in a map.

The *body function* is unique to each API, and data type. This reflects the implementation of these APIs as C++ templates. C++ templates are types *of* types, written as for example `std::list<int>` - meaning a list (`std::list`) of integers (`int`). A declaration such as `std::list<int>` is actually something of a super-powered macro, the expansion of which defines concrete C types and functions.

This compares to containers in dynamic languages, which present the appearance of taking any type of object, but are actually fixed to dealing in a boxed variable type - called the "thingy" or scalar in Perl, ZVal in PHP/Zend, etc. So, it is not possible to provide direct access to the generic APIs without a slow and prohibitively difficult to arrange compilation cycle. Instead, in the Perl bindings, a set of body functions are provided which allow a given API to be used with a particular calling convention and boxed variables. The hope is that the use of boxed variables makes real access to the generic APIs unnecessary.

For example, `threads::tbb::for_int_array_func` is a Perl object type which calls `parallel_for()`, using a divisible range of integers (`tbb::blocked_range<int>` in TBB), a concurrent array for state (in essence, a `tbb::concurrent_vector<thingy>`), and a static function name for use as a callback. The static function is called with two arguments: a grain-sized range, and a reference to the array. More flexible body function types are described in the API section, though currently limited to `parallel_for()`.

2.3 Pipeline Tasks

The `tbb::pipeline` API is for problems where a stream of data is processed through several discrete stages. The first stage of the pipeline produces items to process, like an iterator, returning NULL when it is finished. Intermediate stages transform items, like a `map` function. The last stage need not return anything and is most similar to a simple function call.

Individual processing stages can be stateful, receiving items one at a time, in order, in which case they are marked as serial. Or, they can be stateless, such as always returning the same transformed result for a given input, in which case they are marked as parallel.

Unlike a `map` function, the number of items may not multiply along the way.

The arrangement of stages, and multiplicity selection (serial or parallel) of each stage of the pipeline happens at runtime. Additionally, processes are free to work on any stage of the pipeline which has work to be done, an advantage over approaches in which threads (or processes) are started for each stage of the pipeline.

There is also a `parallel_while` function, which a bit like a pipeline with only an iterator and a final function call. Unlike the pipeline API, the function call also has the capability to add new items to the "pipeline" for processing as if the iterator had returned them. This can be used for instance for genetic algorithms where the result of assessing/scoring a value may indicate more similar values to be assessed.

The TBB pipeline API is yet to be provided by the `threads::tbb` library. Difficulties may arise in that stateful stages of the pipeline may end up thread-bound, and currently there is no mechanism/call for this in the `threads::tbb` module.

2.4 Thread-safe Containers

The TBB library provides several concurrent container template classes: `tbb::concurrent_vector` (an array class), `tbb::concurrent_hash_map` and `tbb::concurrent_queue`. This compares with `threads::lite` which implements only a concurrent queue class.

In a C++ environment, it is not always required to use the concurrent container API to be thread-safe. However in the `threads::tbb` Perl implementation, it is the only way that data from another thread may be accessed.

A system of *lazy cloning* is used to avoid some of the worst-case performance scenarios, to allow the master thread to proceed at "full speed" while worker threads are naturally throttled by the time taken cloning the data structures as they are accessed. This is described in detail in the API documentation.

3 API for Perl 5

Here we will talk about potentially using the TBB API with Perl 5. Much of this section is lifted from the POD of the `threads::tbb` module and other key modules.

3.1 Initializing TBB

The only `threads::tbb` class method is the constructor for a new TBB context. This context is a demand that worker threads have at least the module set specified loaded. By default, workers should end up with the same module set as "now".

```
use threads::tbb;
my $tbb = threads::tbb->new();
```

To make this happen, the library takes a copy of the %INC global variable (see %INC in *perlvar*) at compile time. It also saves and places a special callback onto the @INC global (see require in *perlfunc*) which records all of the modules later loaded by code.

It builds these into two lists which are passed to the worker threads for driving thread initialization before any work is done. They can be specified manually (as in *threads::lite*):

```
my $tbb = threads::tbb->new(
    lib => \@INC,   # default: @INC at module BEGIN time
    modules => [ qw(Math::BigRat) ],
);
```

`lib`

This is an ordered list of paths to prepend to @INC of the worker threads before any modules are loaded. If any paths already exist on @INC of the worker thread, they are not duplicated.

`modules`

This is an ordered list of modules to 'require' in the worker thread. The modules in this list are specified in module-form (eg "Math::BigRat"). If you want to specify instead a list of require-form (eg "Math/BigRat.pm"), this is also possible:

```
my $tbb = threads::tbb->new( requires => [ "Math/BigRat.pm" ] );
```

As the list of modules are processed, if any module encountered is already in the `%INC` - for instance, if it was loaded as a dependency of another module - then it is not re-loaded.

The default is to take the `%INC` saved from the module load, and sort it such that, eg `Moose/Object.pm` sorts after `Moose.pm`, and then after that alphabetically. After this sorted list, any modules which were seen by `require` or `use` are added to the list in the order they were included in the main program.

Note if you add paths to the beginning of `@INC` yourself, *after* `use threads::tbb` but before `threads::tbb->new()`, then `threads::tbb` will not see them. So, put your `use lib "path"` statements before the first `use threads::tbb;`, or specify required modules yourself.

3.2 Parallel APIs

These methods are available on *body objects* which must first be obtained by methods on the `threads::tbb` object.

3.2.1 parallel_for

`parallel_for` can be used to process a set of data. It is passed a **range** object, and a **body** object. The **body** object encapsulates state, and the range selects a part of that state.

You can declare the **body** object using either of the following methods:

`$tbb->for_int_array_func( \@array, "Some::Func" )`

This returns a body object, suitable for use with a **threads::tbb::blocked_int** range, and allows a single `threads::tbb::concurrent::array` for shared state. The `Some::Func` subroutine will be called as:

```
&{"Some::Func"}( $range, $array_ref );
```

If it wants to communicate state, it should do so via the `$array_ref`.

`$tbb->for_int_method( $object, "method" )`

This will create a body object which calls the "method" method of `$object` on sub-divided ranges, as:

```
$object->method( $range );
```

`$object` will be cloned once for each worker, so can be modified and the results expected to stay consistent within the lifetime of the parallel_for; the calling `$object` will see none of them.

3.3 Concurrent container API

3.3.1 Overview

```
use threads::tbb;

# ARRAY tie interface:
tie my @array, "threads::tbb::concurrent::array";
$array[0] = $val;
push @array, @items;

# HASH tie interface:
tie my %hash, "threads::tbb::concurrent::hash";
my $value = $hash{key};  # always deep copies
$hash{key} = $value;    # careful!

# preferred Hash API: for access:
my $hash = tied %hash;          # doesn't need to be tied really
my $slot = $hash->reader($key);
print $slot->get();             # now safe
my $copy = $slot->clone();      # also fine
undef($slot);                   # release lock

# for writing:
$slot = (tied %hash)->writer($key);
$value = $slot->get();          # get the value out
$slot->set([$value]);           # fine
$copy = $slot->clone();         # $copy now a dup of [$value]
undef($slot);                   # release lock

# SCALAR tie interface:
# not really concurrent in any way; and every access may copy in to
# the thread which requests it.  these wrappers for scalars can be
# passed around via the various containers.
tie my $item, "threads::tbb::concurrent::item";
$item = $val;
print $item;
```

The `threads::tbb::concurrent::` series of modules wrap respective tbb concurrent classes. For now there are two main container classes - *threads::tbb::concurrent::array* and *threads::tbb::concurrent::hash*

Note that they are only concurrent if you restrict yourself to the concurrent APIs. Other ways of accessing the containers may result in programs with race conditions.

Also, the SCALAR interface: *threads::tbb::concurrent::item* currently has no locking mechanism, it is currently just an auxilliary way of shunting data between interpreters using the lazy clone method.

3.3.2 Lazy deep copying

The C++ function `clone_other_sv`, from *src/lazy_clone.cc* in the source distribution, exists to implement selecting cloning of data reachable from one interpreter to the next. This is implemented in a lazy fashion.

If entries in the container are requested by a different thread, a deep copy happens then and there, carried out by the worker thread and not the main thread. So long as there is no use of the actual state machine of the foreign interpreter, or side effects on data structures it "owns", this should be relatively safe.

Allowed data types

The initial implementation of the deep copying has very much the same limitations as *threads::shared* - in that only a certain core set of "pure" perl objects can be passed through.

XS objects should be safe - as in, not cause segfaults - so long as the package either defines `CLONE_SKIP` (in which case the objects will be replaced by "undef" in the cloned structure - see *perlmod*), or if they define a `CLONE_REFCNT_inc` method. The `CLONE_REFCNT_inc` method should update the objects' internal idea of how many references are pointing at it, and return the value 42. If it did neither, then the code will emit a warning.

As closures are not supported, inside-out objects cannot be passed - and in fact they'd likely be very inefficient.

Not yet supported are MAD properties or "strange" forms of magic. Overload is currently thought to be safe. Filehandles should be relatively trivial to support but are not implemented yet.

4 Conceptual Findings

4.1 Lazy Deep Copying

Duplicating all data as it is stored into the containers (as in `threads::lite` and `threads::shared`) represents a performance drain in an important case, where the overheads of the duplication overwhelm the useful work.

With lazy deep cloning, if an interpreter requests a value stored by itself, the stored value is returned immediately, so if there is a deep complicated structure behind it then this does not need to be copied.

Without a notion of `const` in the Perl interpreter, and given that side-effects of operations may affect the data structure even when it is merely being accessed (for instance, the hash-internal iterator fields updated by the `each` API), data must in general be duplicated if it moves between interpreters. So, when retrieving from a container, if the slot was stored by another interpreter, then it is cloned by the *retrieving* interpreter. This deep cloning is performed by a custom function which is particular about not modifying the data structure it touches in any way.

The implementation of deep cloning used by the module ends up less restrictive than the Pure Perl version in `threads::shared` about valid types of data allowed through, but more restrictive with magic data structures, which cannot be honoured.

It respects the `CLONE_SKIP` API, requires a reference counting API, but suffers from a deficiency present in the existing clone API, described later.

The advantage to using Lazy Deep Cloning, over an eager algorithm which uses a safe, neutral interpreter that never runs anything (as in *threads::shared*), or to a collection of `Storable::freeze` buffers (as in *threads::lite*) being:

1. reduced memory use; data is only copied to the threads which demand it.

2. there is no overhead for the thread that started the operation to process data; other than that taken receiving completed blocks from workers,

   It is useful if the worst that happens in this case is that the other thread processes a relatively small portion of the data, and so only helps reduce processing times by a small amount, rather than every thread being slowed down all the time.

3. reduced number of overall deep copies. If memory used by a shared interpreter is used instead,

4. likely faster cloning (`clone_other_sv` is implemented in C++ using STL containers, and does not recurse). This is yet to be proven via benchmarking.

5. You can choose to use an eager algorithm by simply `freeze`'ing data on the way in.

   In other words, if it doesn't work out, then it can easily be avoided. The documentation refers to allowing this to be selected via an environment variable, though this is not yet implemented.

The key conceptual finding of this, relevant to designers of interpreters (such as Parrot or novel runtimes for languages such as Python), is that complications arise, and duplications are necessary when the interpreter cannot mark data structures as immutable. If data structures were allowed to be `const`, or a mixture of `const` and concurrency-safe objects, then they could be freely shared between interpreters without duplication.

4.2 Perl 5 CLONE API for XS module writers

As mentioned above, the clone function delivered honours the `CLONE_SKIP` API, which permits classes to opt out of being transferred between interpreters. If the class defines `CLONE_SKIP` as a class function which returns a true value, then any instances of that class will be replaced by `undef` during clone [6].

A new API had to be created to deal with correctly destroying object instances shared between interpreters. This is primarily relevant for XS objects (aka PVMG scalars), in which the IV (Integer Value) slot of the boxed variable holds a pointer. The structure at the target of the pointer must be correctly reference counted, or destruction cannot safely occur. The name `CLONE_REFCNT_inc` was given to this operation to represent the nature of the API and the name for the operation in the Perl 5 C API. Its logical reverse is not `CLONE_REFCNT_dec`, but should be performed during `DESTROY`.

There is still no sensible instance API to allow XS objects to clone state, which applies to `Perl_clone()` as well as the objects passed through the cloning function in this implementation. There is a magic vtable slot, `sv_dup`, which is similar but not flexible enough to permit use outside of `Perl_clone()`. It is also complicated to arrange, requiring the magic API. As `sv_dup()` is barely documented, and an alternative approach outlined in the `perlmod` documentation, going forward it should be possible to replace this API with one which permits use both with `Perl_clone()` and this cloning method.

4.3 Task based vs Event-based programming

The task-oriented model makes an assumption; that each task will be able to perform useful work when it is run and relinquish the processor to other tasks once it is done.

For this reason, it is recommended by the TBB documentation that task body functions should generally be *non-blocking* (that is, not use system calls which must wait for an external slow operation to complete).

Already this points to a similarity between task-oriented programming and event-oriented programming, such as provided by `POE` or `Event.pm`. These libraries will keep internal queues of waiting events, though are limited to only a single CPU resource. Many of the challenges faced by those porting to an event-oriented framework will apply with the parallel, task-oriented approach of the TBB API.

In both event-based programming and task-based parallelism, large amounts of work are converted into queues of ideally short duration function calls. Both systems require re-arranging the program such that the result of a complex computation is handled by a callback, and not implemented inside a function call.

There are key differences, one is that with TBB you may recursively call another parallel function, and expect control to return to that block only when the parallel work is complete.

The similarities point to potential for abstractions which could be useful for running in both purely event-oriented frameworks and concurrent systems like TBB, or potentially even allowing event frameworks to run events over multiple cores.

5 Quantitative Findings

As in the first release, only the `parallel_for` API is implemented, this is the one used for testing.

5.1 parallel_for scalability

5.1.1 a task not particularly well suited

An example script in the `threads::tbb` 0.01 distribution, `examples/incredible-threadable.pl`, can be used to demonstrate both the effects of memory allocation overhead, and show the case for lazy cloning as a strategy.

The problem is prepending a string with "Ex-", and appending a small token to mention which thread processed the string. This entirely nonsense task places the emphasis on the overheads.

This first result demonstrates that a speedup is possible; it does not take into account the overhead of using the API in the first place, just that the single-threaded use is slower than the two-threaded use. The input is a 1.5MB, 46k line file, and the machine sports a 2GHz Core(TM)2 T7200 CPU (with two physical cores and two threads). Median of five runs.

Threads	Processing Time	Wallclock Time	Master Count	Worker Count
1	330ms	0.74s	46332	-
2	274ms	0.67s	31129	15203

The main thread here was able to process data at approximately 2-3 times the speed of the workers. For this problem, simply copying the data in and out is the major overhead, and so it helps that the master thread can proceed at "full steam" for avoiding the overheads exceeding the return, such that the wall clock time is lower than the single-threaded case.

Due to the design of the implementation, relatively large runs of data must be used to see results. This is because the processing time actually includes the amount of time it takes to start the worker Perl interpreters and load modules, which on these systems starts at around 30ms per core, though it will happen in parallel. However in long-running programs this startup penalty will be amortised.

By running on a larger system with increasing numbers of threads we can see if this speedup continues as more processors are added, as we would expect. An Amazon EC2 instance with 8 virtual cores was used for the following testing.

Threads	Processing Time	Master Count	Worker Count
1	371ms	71063	-
2	270ms	44414	26649
4	191ms	29147	41916
8	263ms	26651	44412

A scaling issue is evident in these results. Up to 4 cores, the processing time decreases, but then takes a sharp decrease.

By re-running the test with the TBB scalable memory allocator, the speed-up continues to 8 cores:

Threads	Processing Time	Master Count	Worker Count
1	370ms	71063	-
2	263ms	44414	26649
4	155ms	28150	42913
8	125ms	21426	49637

If this were a real program, getting a 3x speed-up from 8x the processing power would probably not be considered a very good result, but here the purpose is to demonstrate the need for scalable memory allocation as cores increase.

The actual program demonstrated may not resemble a real-world application, but the scalability issue is caused by operations all threads must perform to do any work - allocating and freeing memory. In the Perl interpreter, entering and exiting functions involves allocating and freeing memory from the heap, not the C stack (a more efficient operation). Additionally, many Perl opcodes will involve memory allocations, for instance string manipulation. All of these allocation operations may be protected by a single mutex, making it a heavy contention point.

Furthermore, given that the requirements of memory allocation by the interpreter is not specific to the implementation, the findings may be relevant to other multi-threaded uses of Perl.

5.1.2 A synthetic task

In this section, the above test is repeated, but replacing the trivial, do-nothing worker function with a simple loop that counts a fixed number of times. That is, the inner loop becomes:

```
for (my $i = 0; $i < $num; $i++) { }
```

Where `$num` is the grain size. The total number of increments is referred to as the "Set size".

The purpose is to discover how much work should be performed in the body function for the program to be scalable.

We follow the procedure in [7] - first, in single-threaded mode, find a grain size which is around 5-10% extra work. Single runs (if a larger grain size resulted in a slower time than a previous time, then the result was repeated - this is possibly an artefact of the paravirtualized test platform)

Threads	Set size	Grain Size	Time	Rate
1	1M	1	5.6s	179 kloops
1	1M	2	2.75s	255 kloops
1	1M	4	1.45s	688 kloops
1	1M	8	762ms	1.31 Mloops
1	1M	16	428ms	2.34 Mloops
1	10M	32	2.55s	3.92 Mloops
1	10M	64	1.91s	5.24 Mloops
1	10M	128	1.49s	6.71 Mloops
1	10M	256	1.08s	9.29 Mloops
1	10M	512	955ms	10.5 Mloops
1	10M	1024	916ms	10.9 Mloops
1	10M	2048	892ms	11.2 Mloops
1	10M	4096	879ms	11.3 Mloops
1	10M	8192	867ms	11.5 Mloops
1	10M	16384	859ms	11.6 Mloops
1	10M	32768	859ms	11.6 Mloops
1	10M	65536	859ms	11.6 Mloops

This initial testing indicates that a grain size of 512 or greater should have only a ~10% impact on uni-processor, or around 2048 for a stricter 5% allowance.

So, now we run with a number of threads and grain sizes in the 512-2048 range; average of best three runs out of five (to counteract effects of paravirtualized platform):

Threads	Set size	Grain Size	Time	Rate	Efficiency
1	16Mi	512	1.71s	9.8 Mloops	-
2	16Mi	512	870ms	19.3 Mloops	98.3%
3	16Mi	512	682ms	24.6 Mloops	83.5%
4	16Mi	512	520ms	32.2 Mloops	82.2%
6	16Mi	512	426ms	39.3 Mloops	66.8%
8	16Mi	512	388ms	43.2 Mloops	55.1%
1	16Mi	1024	1.54s	10.9 Mloops	-
4	16Mi	1024	464ms	36.1 Mloops	83.1%
8	16Mi	1024	361ms	46.5 Mloops	53.4%
1	16Mi	2048	1.48s	11.3 Mloops	-
4	16Mi	2048	440ms	38.1 Mloops	84.3%
8	16Mi	2048	459ms	36.5 Mloops	40.4%
1	64Mi	2048	5.98s	11.2 Mloops	-
2	64Mi	2048	3.04s	22.1 Mloops	98.4%
4	64Mi	2048	1.78s	37.6 Mloops	83.9%
8	64Mi	2048	1.01s	66.4 Mloops	73.9%

The above results were all obtained without the scalable memory allocator.

The last column, “Efficiency”, represents the throughput improvement compared to the extra number of CPUs which were made available. It is simply calculated as the time for 1 Thread execution for the given Set size and grain size, divided by the time for this run, divided by the number of threads.

With the scalable allocator, the efficiency can be even higher:

Threads	Set size	Grain Size	Time	Rate	Efficiency
1	16Mi	512	1.62s	10.4 Mloops	-
2	16Mi	512	878ms	19.1 Mloops	92.3%
3	16Mi	512	584ms	28.7 Mloops	92.5%
4	16Mi	512	455ms	36.9 Mloops	89.0%
6	16Mi	512	338ms	49.6 Mloops	79.9%
8	16Mi	512	273ms	61.5 Mloops	74.1%
1	16Mi	1024	1.53s	11.0 Mloops	-
2	16Mi	1024	826ms	20.3 Mloops	92.6%
4	16Mi	1024	426ms	39.4 Mloops	89.8%
8	16Mi	1024	258ms	65.0 Mloops	74.1%
1	16Mi	2048	1.50s	11.2 Mloops	-
4	16Mi	2048	425ms	39.5 Mloops	88.2%
8	16Mi	2048	424ms	39.6 Mloops	44.2%
1	64Mi	2048	5.98s	11.2 Mloops	-
2	64Mi	2048	3.09s	21.7 Mloops	96.8%
4	64Mi	2048	1.58s	42.5 Mloops	94.6%
8	64Mi	2048	854ms	78.6 Mloops	87.5%

As expected, the most impressive results are with the largest grain size, the largest input set and the scalable memory allocator.

This data point yielded the highest overall rate and efficiency. 87.5% scalability over 8 processors is scaling to 7 processors out of 8 and a good preliminary result, even for a synthetic test, especially given the paravirtualized platform.

Improving this further would likely require use of a dedicated highly parallel platform, as well as eliminating platform inefficiencies such as kernel-level issues [9].

There are some outlying data points, even with the "average of three best runs over 5" paravirtualization compensation. So the results should be read into with caution.

Nonetheless we should be able to see the overhead as the grain size gets smaller and smaller, confirming 512-2048 as reasonable runloop iteration counts for Perl programs.

Threads	Set size	Grain Size	Time	Rate	Efficiency
1	16Mi	256	1.80s	9.4 Mloops	-
4	16Mi	256	520ms	32.3 Mloops	86.2%
8	16Mi	256	277ms	60.6 Mloops	80.8%
1	16Mi	128	2.13s	7.9 Mloops	-
4	16Mi	128	616ms	27.3 Mloops	86.4%
8	16Mi	128	350ms	47.9 Mloops	76.1%

While the relative speedup is still in the 75% range for 8 cores, the overall rate is so far diminished by that point that in a real sense the overheads are still swamping useful work.

5.1.3 A practical application

In this section, a real-world program is taken and the parallelism retro-fitted. The program chosen was `album`, a web gallery program. This program spends most of its time resizing images. For the sake of the test, the *convert(1)* program was recompiled to not use OpenMP. This of course makes this result highly synthetic, though the script itself is real-world.

The system was another Amazon EC2 xlarge instance, and the gallery being thumbnailed was approximately 118MB in size. All figures average of five runs.

Threads	User Time	Real Time	System Time
1	43.0s	49.3s	4.7s
8	46.9s	8.8s	8.8s

The speedup for the whole program run is not 8 times on an 8-way system, but the image resizing phase of the program operation is visibly much faster; the overall speed-up is 5.6x. Not bad for changing 43 lines in a script almost 8,000 lines long!

6 Closing notes

Firstly, it should be emphasised that retrofitting threading to an interpreter is a task not to be taken on lightly. It must be remembered that an interpreter is a form of virtual machine, and that its primary task is to move the state of the virtual machine forward. It is only natural that the action taken from each state is intrinsically linked to the previous state. Unless the language being interpreted provides some concurrent APIs, there will be nothing for worker threads to do.

Secondly, while the TBB API has provided convenient scalable algorithms, the mechanics of connecting those to actual interpreter instances, and dealing with shared state are fundamentally challenging. The work here is in very many ways standing on the shoulders of the pumpkings that have worked on Perl threading over the many years of its life. It has been carried out with ready access to existing implementations, and with extensive reference to source code history and comprehensive internals API documentation.

It is tempting to use the results of this to infer that a similar effort could be undertaken against the PHP/Zend interpreter. Starting slave interpreters in threads could be integrated using documented APIs as a SAPI interface, and mechanisms for passing data around implemented corresponding to the findings in this effort. However the very real practical obstacles that will arise as a result of there having been no serious effort from the core team to implement any kind of threading, and the lack of internals documentation, will make this a task that very few will be able to take on.

Thirdly, there may be serious mismatches in expectations of a library written for one language and what the other language is providing. There is no compelling reason that interpreters should not be able to share pointers to shared state, but because the expectation is violated, much copying of data is required and this places another overhead that makes scaling programs difficult.

On the positive side of things, there are some good results demonstrated for tasks which suit parallelisation, where the API is available. While some potential pitfalls have been identified, largely speaking binding new parts of the TBB API to Perl has progressed successfully. So, the further development of this module to cover the rest of the TBB core APIs - reduce, sorting, pipeline programs, and the raw task API - is now a relatively known quantity.

References

[1] The Perl Source History, Wall et al, `http://perl5.git.perl.org/perl.git`

[2] Based on conversations with two past pumpkings, IRC and oral.

[3] threads::lite, Leon Timmermans, `http://search.cpan.org/dist/threads-lite`

[4] Getting Started with Erlang User's Guide Version 5.8.3, Chapter 3, "Concurrent Programming", section 2 "Message Passing", `http://www.erlang.org/doc/getting_started/conc_prog.html#id64727`

[5] This figure is based on results covered later, which admittedly were not performed using `threads::lite` itself, and naturally the actual figure observed will depend on the program being tested.

[6] , Perl 5 core documentation, "perlmod" - Perl modules (packages and symbol tables), under section "Making your module threadsafe"

[7] James Reinders, *Intel Threading Building Blocks*, O'Reilly. ISBN-13: 978-0-596-51480-8

[8] Patch series to thread `album` using `threads::tbb`, `https://github.com/openparallel/album/compare/5d8bda760a...cbd2e8d7bb` `http://xrl.us/opalbum`

[9] Theodore T'so gave an interesting talk at Linux Conference Australia 2011 in Brisbane on this topic; one key point made was that since the Linux Scalability Effort concluded, little serious effort has been put into making Linux ultra-scalable across highly multi-core systems. That is, the tested platform, Linux 2.6.35 running Ubuntu maverick 64-bit, may itself only scale to 7 out of 8 processors.