Optimizing Web Performance with TBB

Open Parallel Ltd
lenz.gschwendtner@openparallel.com

August 11, 2011

Open Parallel is a research and development company that focuses on parallel programming and multicore development. We are a bunch of highly skilled geeks from various backgrounds that work together on problems in parallel programming and software development for multicore and manycore platforms.

At LinuxConf 2010 in Wellington James Reinders gave a talk about the Threading Building Blocks (TBB) library, a C++ threading library that sets out to make multicore programming more accessible to the average programmer. We took this idea on board and explored the possibilities of opening up this approach to an even wider audience, namely the audience of web application developers working in script languages.

This paper explains our success and the path that lead up to it. We explain the work we did in Perl and PHP, how we integrated TBB functionality into WordPress by running WordPress on HipHop and in the end we will briefly show you the demos we wrote.

The demos are small Perl and PHP projects that can be played with at `http://geopic.me` and the source code can be downloaded from `http://github.com`.

All code we wrote in this project is free software that can be downloaded from our github repositories. The respective links are in Section 5.

Contents

1 Scripting Languages

1.1 Why scripting languages

One of the largest uses of custom written software today is some sort of web enabled software. This huge part of the software development industry is nowadays mainly backed by scripting languages. Even huge sites like Facebook, Yahoo! and Twitter are mainly written in scripting languages as it is faster to develop and often easier to deploy. The top three scripting languages are Javascript, PHP and Perl. With the lack of direct access to a CPU in Javascript we set out to look at PHP and Perl.

1.2 Why parallel

Many websites require a non-trivial amount of per-request processing in the application layer, perhaps to retrieve, consolidate or otherwise manipulate data. Achieving better performance at this level improves response times and the overall user experience. Even when processing time at application level is not critical, parallelizing access to database and web service backend layers can yield substantial improvements in perceived performance.

With many scripting languages writing non-linear code is either really hard or near to impossible. PHP lacks any sort of parallel execution that does not require an entire new PHP process. Perl offers more in this area but it is still considered a non trivial art.

This drove our goal of adding TBB support into PHP and Perl, starting with HipHop as the PHP implementation of choice and later on adding Perl support to the game.

1.3 Implementation

Implementing TBB into many current language interpreters is a challenging problem, due to both a lack of basic concurrency support in those interpreters (and sometimes in the language syntax), coupled with a *disconnect* between the architecture of TBB and the architecture of the interpreter.

There are one failed and one not widely used threading approach in Perl and every attempt in PHP to implement threading has failed so far. The core developers on both sides are very much in doubt if it is a path worth going down at all. The problem is global locking and copying/sharing of data structures that are thread local. Our Perl implementation is a starting point that could influence not only the Perl community but other language designers and interpreter developers as well.

In the Perl community we are trying to lobby for a const keyword that would lock a data structure and remove the need to copy it into every thread. The ability to make something immutable is missing in Perl and PHP and this makes the startup cost of any worker thread very expensive. For our Perl library we wrote a lazy clone module that would only clone a data structure if the worker thread really accesses it. That way we only penalize the worker thread for accessing data - we can possibly get around cloning structures at all if they are not accessed within this task.

On the PHP side we used HipHop which is already thread safe and extended it with TBB data structures and library functionality. This task was easier in that respect as we already had a very well designed compiler that only needed extending.

1.4 Lightweight co-routines

Another problem we tried to solve on the way is the problem of executing tasks in parallel that may not be the same tasks (no `for` loop) like fetching content via HTTP while resizing an image. In PHP this can only be done in serial or by using a third party tool like Gearman. With TBB we can run multiple threads in parallel and could run those tasks in parallel on several CPUs without the need of spinning up an entire web server instance. This looked intriguing and we got it working in the end.

1.5 The goal

We managed to get TBB support into Perl and PHP and managed to expose some of the TBB functionality into scripting languages. This major goal of ours was reached with the work we explain in Section 3 and Section 4. Our goal of exposing a simple threading concept in scripting languages was reached and we think we added a significant piece of work and research to both, the Perl and the PHP community. We will continue to work on these topics and will continue to expose threading concepts to script languages. Our work is not over yet but we are a big step forward.

2 geopic.me

Before diving into the implementation details I want to show two little tools as real world demos and as working code to look at. The demos are based around the HTML5 geo tag which is present in modern browsers and can be read with a Javascript API. Both demos run on a Ubuntu 10.04 LTS server with four emulated CPUs hosted by Rackspace. They run behind a nginx web server to put the thinnest layer of abstraction on top of the actual code. The demos are both, fun to look at and interesting to read the code.

2.1 PHP version

The PHP version is running on HipHop and is called via AJAX calls in a static HTML page. The page itself is a simple HTML page with only as much Javascript as we need to call the HipHop code via AJAX.

2.2 What we demonstrate

In the HipHop version we read the current Lat/Lon from the accessing browser via the geo tag and then search Twitter for tweets of that area. We then filter out all tweets that have image URLs in it and in parallel search Flickr for recent additions in the same area. The demo is mainly about fetching information in parallel and shows as a result a stream of images of your area. When looking at the code base we can easily track where things happen in parallel which would normally require a separate web server instance or some sort of third party tool.

2.3 Setup

To set up the HipHop demo we expect that you have prior knowledge of how HipHop works. If you don't then have a look at the HipHop Wiki [1] which explains all the steps necessary to get HipHop working on different platforms. When starting with HipHop keep in mind that you actually set up a build tool chain that takes your PHP code, turns it into C++ and then compiles it down to a Unix binary. The Unix binary can then be run from the command line and is in fact a small web server running your translated PHP code.

[1] https://github.com/facebook/hiphop-php/wiki

For the HipHop version to work you need to clone our HipHop version - with our patches, as compared to the official Facebook version - from Github which can be found on github.com [2].

Compile HipHop the same way as described in the Facebook Wiki or use your existing HipHop build environment only with our repository instead of the Facebook one.

When you have a working HipHop you need to clone our geopic.me code from github.com [3].

Compile it down to a binary either with your preferred options or with the following command line (assuming you set up the environment variables as suggested in the tutorial).

```
# $HPHP_HOME/src/hphp/hphp --input-list=files.list -k 1 --log=3
```

Note the temp directory HipHop compiled into and export it to `$BASE`, then run the demo with:

```
# ${BASE}/program -m server --p 8082 -v Log.Level=Verbose
```

You can now point your browser to port 8082 [4] and play with the demo. Also watch your terminal that started the demo for some debug output.

The nginx config snippet that would make the demo work and map all the right paths looks like this:

```
location / {
    root   /var/www/geopic.me;
    index  index.html;
}

location /hphp/ {
    proxy_pass   http://127.0.0.1:8082;
}
```

[2] https://github.com/openparallel/hiphop-php
[3] https://github.com/openparallel/geoPic
[4] http://localhost:8082

2.4 Perl version

The Perl version is based on Mojolicious - a micro web framework that is written to fill the gap in web frameworks for small projects like this.

2.5 What we demonstrate

In the Perl demo we search Flickr for photos of the area based on the geo tag. We then fetch all the images and cache them locally. We then offer two different scaling methods, one running single threaded and one with TBB. Both methods take the cached images, scale each image into a smaller one and then compose one big image out of them. The single threaded process takes on average about a second longer than the TBB based process. Interesting to note is also that the TBB based version has a startup cost for the first iteration and then can play out its strengths in every iteration after that first one.

Under the image grid is a small stats table that gathers stats about the different phases of scaling and assembling of the image and calculates some averages for both scaling methods if they are run more than once.

2.6 Setup

The Perl version needs the ImageMagick libraries and TBB, ImageMagick normally ships with some OpenMP optimizations that can change the characteristics of the tests a bit. If you try to only run the demos for getting to know `threads::tbb` then you can just install the necessary libraries. If you are interested in replicating our demo setup and making sure that ImageMagick is doing the right thing than you have to compile it without OpenMP support. On a Debian based system this may look like this (no guarantees for success and best done in a controlled environment like a chroot or a VM)

```
sudo apt-get build-dep imagemagick
sudo apt-get install build-essential fakeroot
apt-get source imagemagick
cd imagemagick-*
vi debian/rules     # add --disable-openmp to the 'configure'
command-line
dpkg-buildpackage -b -uc -rfakeroot
cd ..
dpkg -i *.deb       # install the freshly built packages
```

Furthermore we need the TBB library:

```
sudo apt-get install libtbb-dev
```

And finally we need a set of Perl libraries:

- `JSON::XS`
- `LWP::Simple`
- `Mojolicious::Lite`
- `Flickr::API2::Cached`
- `threads::tbb`

All that is missing now is the actual project which can simply be cloned from github.com or downloaded from our github repository [5]

To run the demo simply change into the directory and fire it up with the following command line:

```
# ./geopicme-pl daemon
```

This starts the demo on port 3000 and you can point your browser to localhost [6] and play with it. Also make sure to have a look at the terminal that started the demo for some informative debug output.

The nginx config snippet that would make the demo work and map all the right paths looks like this:

```
location / {
    root   /var/www/geopic.me;
    index  index.html;
}

location /perl/ {
    proxy_pass   http://127.0.0.1:3000;
}
```

[5] https://github.com/openparallel/geoPic-pl
[6] http://localhost:3000

3 HipHop

HipHop is a PHP to C++ cross compiler that was developed by Facebook to cut down on resource needs and speed up the execution times of their gigantic web infrastructure that was started on a classic PHP/MySQL stack and now has to scale to hundreds of millions of users. The HipHop project is a PHP implementation that is thread safe and already uses TBB for some memory management.

3.1 Implementation

We started by extending the existing support and added first only the new **parallel_for** function. Later, we added concurrent data structures and re-implemented our first approach. What we have now is a robust implementation of **parallel_for** and **parallel_reduce** with the data structures needed to support them.

What we learned on the way was both, very enlightening and quite frustrating at times. Our aim to make TBB more widely accessible was reached by getting the language extension into HipHop but we also tried to get it into Zend PHP. This turned out to only work with a language compatibility module that does not provide the functionality we can offer on the HipHop platform. The reason for this is the architecture of the PHP interpreter - PHP is designed to run a single thread per web request and provides no support for multiple threads at this level. Because of this, we provide a version of the module for PHP that emulates the TBB calls using a single thread. This allows a single codebase to be used in both TBB and HipHop - obviously the speed-ups from parallelism only happen in HipHop.

3.2 WordPress

In our work with the PHP HipHop compiler we also wrote a patch set for WordPress and enhanced WordPress with our new **parallel_for** language extension. This trial brought us instant success in reduced page load times. The patch set for WordPress only replaced some key **foreach** loops with **parallel_for** and was our first real success with the TBB library in PHP. Based on that success we started out to re-implement our initial approach and tidy up our patch set for HipHop to make it more accessible to others.

3.3 Who benefits from it

PHP applications that benefit from TBB typically perform work in responding to user requests that is CPU, resource or latency bound. This could be a complex

mathematical transform, a collection of database queries or web requests. Using TBB, a traditional iterative operation is converted to use a **parallel_for** or **parallel_reduce** idiom. This relatively simple code change enables the application to take advantage of multi-threading. There is no need for the developer to be concerned with traditional synchronization idioms or to worry about deadlock and other concurrency issues.

3.4 Example

To illustrate what a code change looks like when rewriting iterative code into parallel code we will give you a short example.

This is a simple iteration through an array:

```
for($n = 1; $n<$array.count(); $n++) {
   process($n, $n+1, $array);
   // process(a,b,c) manipulates the slice
   // of the array 'c' from 'a' to 'b'-1
 }
```

This can be rewritten using TBB as:

```
parallel_for_array($array, 'process');
```

In this version `process` will be called in parallel using TBB calls and now runs optimized to the underlying CPU architecture.

This demo shows that the code change needed is pretty small but the speedup can be dramatic.

4 threads::tbb

The Perl project worked towards a Perl module that can be used to get access to TBB functions directly. We also started out to implement the core memory structures and then built on top of those the **parallel_for** functionality. The module we have now is stable enough to demonstrate the gains we can get by using TBB in Perl.

4.1 Architecture

With `threads::tbb`, you do not write your algorithms from the perspective of a thread and what the thread should do next. Instead, a selection of parallelism primitives which have been found to be workable and scalable are provided.

Just as when writing *co-operative multi-threading* programs as with Event or POE, the challenge is to break heavy work into small but substantial, generally non-blocking chunks of work. *Substantial* is yet to be quantified; it's likely to be around the ballpark of 1,000's of Perl runloop iterations. Unlike event-based programming, you can freely recurse into the library, start new parallel sections, and expect all runnable tasks to process, up to the number of threads that you started.

As your program runs, the API allows the TBB library to keep queues (trees actually) of runnable tasks. These are identified and kept in thread-affinitive task lists. Other threads can come along and *steal* work from these lists, to keep cores busy.

What this means is that it is relatively easy to make programs which can make best use of processing power available on newer multi-core CPUs. It also avoids per-thread overheads, to only start as many threads are required to use all of the parallelism in hardware. Each thread requires its own C stack and complete Perl interpreter. Therefore it is generally not desirable to create more threads than the hardware has available.

When the first `threads::tbb::init` object is made, one worker thread is created for each processor core or virtual core. This is performed by the TBB library before the Perl interpreter can use strict;

Subsequent calls to it will not create new worker pthreads, instead they will re-use the existing threads.

Each worker thread is for the most part, completely isolated from the other threads - just like `use threads`. Unlike `use threads`, the `perl_clone()` function is never used. Instead, each interpreter must load all of the modules required to get it to do useful work on its own. This is largely automatic, however it is not foolproof and you will benefit from using the constructor thoughtfully.

Worker threads do not share any Perl variables with the main process. A system of *lazy deep cloning* is used to transport Perl data structures between threads; you must pass data through these objects, as they are the only objects which are the same between threads

4.2 Code in the wild

We are only starting out to use `threads::tbb` in the wild but we have used it in to retrofit `album` with parallelism. This program spends most of its time resizing images. The image resizing problem is a common one and many websites that have thumbnails know it. We took `album` which used to work sequentially and turned it into a parallel script using TBB. We did many tests with it and tried to figure out how to tune TBB for optimal use in Perl.

We tested on an Amazon EC2 xlarge instance with a gallery being thumbnailed of approximately 118MB in size. The speedup for the whole program run was not 8 times on this 8-way system, but the image resizing phase of the program operation was much faster; the overall speed-up is 5.6x. Not bad for changing 43 lines in a script almost 8,000 lines long

4.3 Example

To illustrate what `threads::tbb` code looks like and how one would go about rewriting code with `threads::tbb` we will take the example from earlier only in Perl this time.

This is the simple iteration through an array from before only in Perl:

```
foreach $n (@array) {
   push(@result, process($n));
}
```

This can be rewritten using threads::tbb as:

```
our $tbb = threads::tbb->new( requires => [ $0 ] )
    unless $threads::tbb::worker;
@result = $tbb->map_list_func("main::process", @array);
```

5 Further reading

To sum up our experience with TBB and script languages we know now that threading interpreters buries its very own set of challenges but we were able to get further than others did on the same mission by using TBB. The libraries we produced so far - which are open source and can be found on our github account [7] - will be further developed and maintained. We will continue working on both platforms to expose the power of multicore CPUs to developers in an approachable way. The approach of TBB to expose multi threading to developers in a problem oriented way instead of a pure thread oriented way seems like a very natural way of retrofitting threading to existing problems. We highly recommend reading the TBB book (Reinders [2007]) that James Reinders wrote even if you are not a C++ developer as it is very relevant for the work we did and is also a very good way to learn more about threaded programming in general.

Along the way we also produced a number of more detailed white papers covering various aspects of the project:

- **theads::tbb** (Vilain) A paper about the Perl library
- **TBB in WordPress** (Gschwendtner) A paper about our work with TBB on HipHop and in WordPress
- **WordPress on HipHop** (Gschwendtner) A paper about porting WordPress to HipHop

Get in touch if you are interested in these projects or have questions about the work we did. There is further information on our website on `http://openparallel.com`.

5.1 Contacts

general Nicolas Erdody: nicolas.erdody@openparallel.com

technical Lenz Gschwendtner: lenz.gschwendtner@openparallel.com

[7] https://github.com/openparallel

References

James Reinders. *Intel Threading Building Blocks, Outfitting C++ for Multi-core Processor Parallelism*. O'Reilly Media, July 2007.

Sam Vilain. Threading perl using TBB: the CPAN module and white paper, May 2011. URL `http://openparallel.com/2011/05/11/threading-perl-using-tbb-the-cpan-module-and-white-paper`.

Lenz Gschwendtner. TBB in WordPress, October 2010a. URL `http://openparallel.files.wordpress.com/2010/09/tbb-in-wordpress-oct-10.pdf`.

Lenz Gschwendtner. WordPress on HipHop, November 2010b. URL `http://openparallel.files.wordpress.com/2010/09/wordpress-on-hiphop-nov-10.pdf`.

Intel. Threading building blocks. URL `http://threadingbuildingblocks.org`.

GitHub. Open parallel git repositories. URL `https://github.com/openparallel`.

Open Parallel. Open parallel website. URL `http://openparallel.com`.