

The Limits of Concurrent Read-Only Access in the Perl C API Using OpenMP

Brett Estrade

Acutis Computer & Data Consultancy¹

Science Perl Committee², The Perl Community

Project Lead, The Perl+OpenMP Project

Houston, Texas, USA

September 8, 2026

Abstract

This paper investigates the limits of concurrent read-only access to the Perl C API using a suite of Perl modules that makes it straightforward to inline C code with OpenMP and control the OpenMP run-time environment. The Perl C API provides no general guarantee of thread-safety; nevertheless, particular operations can behave safely under constrained access patterns in which multiple native threads read the same established Perl data structure without triggering interpreter-managed mutation. The experiments reported here examine such patterns for scalars (SV), arrays (AV), and hashes (HV) using Perl 5.40.0 built with GCC 12.2.0 and the GNU `libgomp` run-time. Special attention is given to operations such as `av_fetch` and `hv_fetch`, whose safety depends on using non-mutating modes and avoiding structural changes such as AV growth or HV rehashing. The practical objective is to determine where Perl data can be read directly from OpenMP-enabled C code and where conversion, synchronization, or a different interface remains necessary. The results form part of the broader Perl+OpenMP Project effort, which has two complementary aims: provide practical CPAN tooling for using OpenMP from Perl today, and use that working environment to catalogue useful Perl C API operations for shared-memory native extensions while motivating explicitly specified concurrent-read interfaces for operations whose current behavior is sensitive to shared state, including hash traversal. Later improvements to the project toolchain are discussed separately and do not alter the Perl, GCC, or OpenMP run-time versions used for the experiments reported here.

1 Introduction

The Perl language is famously called the “*Swiss Army Chainsaw*,” a term that highlights its greatest strength: the seamless integration of multiple programming paradigms. With Perl, one may elegantly combine procedural code (like C), functional programming (like Lisp), and object-oriented principles (like Java) into a single coherent script. Beyond these paradigms, Perl excels at incorporating elements from the Unix command line, shell scripting, and POSIX system call interfaces. Its true superpower lies in how it enables programmers to work efficiently by *getting out of the way* for those skilled in its use. This latter quality of Perl is the goal when anyone speaks of *idiomatic Perl*.

However, Perl was created during an era when most computers available for general use had only a single CPU. One may confidently state that Perl is *uniprocess*³ to its core, literally. The idiomatic way to *parallelize* Perl, and really all Unix *userland* programs back then, was to leverage the operating system’s ability to clone or `fork` a process. This is how Perl programmers, both then and now, are able to share work across multiple `perl` processes. Many use `fork` directly, while others prefer modules like `Parallel::ForkManager`⁴ to make it easier to manage the `fork`’ing with limits. It also provides event callbacks and basic mechanisms for child processes to communicate results back to the parent. However, this type of standard inter-process communication (IPC) is extremely ineffective for many situations where what one really wants is shared memory over which cooperating threads of execution may implicitly communicate address (or memory location) state. Although there are other IPC options and ways of distributing work in Perl, none escape the constraints of having to use multiple `perl` processes and deal with the challenges of communicating among these full-fledged operating system processes. This includes a popular and well done framework for work sharing, called the *Many-Core Engine for Perl* (MCE)⁵. In reality, it is a *many-process engine for perl* that facilitates work sharing and IPC; using Perl+OpenMP Project modules with each one of those `perl` processes offers some interesting hybrid execution models that may be worth exploring one day.

The genesis period of Perl was gracefully untouched by conversations about shared memory, thread-safety, or race conditions - factors largely irrelevant in simpler times; so `perl` leveraging its serial nature to make memory access and execution fast was never really seen as a compromise. Nearly 40 years later, it is now a liability, multi-core SMP

¹A DBA of Coastal Computing Services, LLC. Correspondence: brett@acutisdata.com.

²<https://science.perlcommunity.org>

³i.e., sequential or single process; as opposed to *multiprocess* or *concurrent*

⁴<https://metacpan.org/pod/Parallel::ForkManager>

⁵<https://metacpan.org/dist/MCE/view/lib/MCE.pod>

now being an extremely common operating environment (that all major operating systems dutifully support). The exception where SMP machines are not the de facto environment, might be inexpensive single-core virtual servers hosted in the *cloud*⁶. Relegating Perl scripts to these environments for system maintenance tasks, has unfortunately contributed to the perception that Perl is not a language to be used for serious and sophisticated tasks; particularly in environments with *many* real CPU cores sitting idle, waiting to be utilized. Perl programmers know, of course that this is completely false.

The concept of *threading* in Perl is also not new, and this paper will not delve into the many ingenious solutions Perl developers have devised over the years⁷. Instead, it highlights an opportunity: the potential to leverage existing compiler technology to introduce process-level, shared-memory threads into a Perl programmer’s toolbox at their own pace. OpenMP is not a *silver bullet*[1] replacement for any of the current solutions in Perl for high level *threading*, but it is a fundamental technology that puts lower-level shared-memory threading into the hands of *journeymen* Perl programmers. Moreover, familiarity and trust in OpenMP as a compiler technology has the potential to influence the Perl run-time itself, opening doors to capabilities unavailable to other languages. By judiciously applying OpenMP today, and by using that experience to identify precise functions in the Perl core and Perl API that could eventually receive explicitly documented concurrent semantics, Perl can mature further on multi-core systems without discarding the sequential assumptions that have long contributed to its simplicity and performance.

This philosophy underpins the **Perl+OpenMP** Project. The project has a practical track and an investigative track. The practical track packages the compiler, run-time, environment, and **Inline::C** integration needed to make OpenMP usable from ordinary Perl programs. The investigative track uses that environment to determine where the boundary between interpreter-owned state and native parallel work can safely be drawn, and to advocate narrowly scoped thread-safe Perl API interfaces where the evidence supports them. The current project tooling does not make the Perl C API generally thread-safe, and this paper does not claim that it does. Rather, the tooling makes the experiments reproducible and gives Perl programmers useful OpenMP facilities now, while the API investigation asks what Perl itself might eventually guarantee more explicitly. OpenMP has been available through GCC since 2007, so there has been ample time for this combination to mature from isolated experiments into reusable Perl infrastructure.

2 A Survey of GCC’s OpenMP Support

gcc has supported OpenMP since 2007, via the GOMP project[2]. This translates to GCC Version 4.2. During this same time period Perl 5.8.8 was the latest Perl release. This means that, as of the writing of this paper (2025), OpenMP has been *trivially* available to Perl programmers for over 18 years! Although a precise deployment census is outside the scope of this paper, GCC has long been a common compiler for Perl on Unix-like systems, so OpenMP-capable toolchains have been available to a substantial portion of Perl installations for many years. For historical context of the sheer magnitude of lost opportunity cost for Perl, Table 1 summarizes when common Linux distributions provided easy access to GCC 4.2. Table 2 illustrates the consistent progress GCC has made in supporting modern OpenMP standards. And to underscore the point that OpenMP is a stable technology with widespread and long-term support across platforms and industry, Table 3 lists representative native compilers capable of building Perl together with selected milestones in their OpenMP support. The opportunity to combine Perl with OpenMP is therefore not historically unique to GCC, even though GCC is the compiler family exercised in this work.

The experimental results in this paper are specifically grounded in a Perl built with GCC and in the GNU **libgomp** run-time. At the time of those experiments, that was also the principal configuration targeted by the **Perl+OpenMP** Project toolchain. The project has since improved its portability. In particular, **Alien::OpenMP** 0.3.10 added compiler-family detection based on the compiler configured into the running **perl**, explicit regression coverage for Linux GCC and Clang, Strawberry Perl on Windows, Apple Clang on macOS, and FreeBSD, together with a GCC 10 through GCC 16 CI sweep.⁸ These are subsequent improvements to the project infrastructure; they do not broaden the experimental claims in this paper, which remain tied to Perl 5.40.0, GCC 12.2.0, and GNU **libgomp**. Additional compiler and platform requests remain welcome through the **Perl+OpenMP** Project GitHub page.⁹

Unless otherwise noted, the experiments and code paths discussed in this paper were exercised using Perl 5.40.0 built with GCC 12.2.0 and the GNU **libgomp** OpenMP run-time. This distinction matters when discussing OpenMP versions: GCC 12 provides complete OpenMP 4.5 support for C and C++, substantial but incomplete OpenMP 5.0 support, and selected OpenMP 5.1 features[3, 2]. Accordingly, this paper does not assume complete implementation of OpenMP 5.0, 5.1, or 5.2 merely because features from those specifications are present in GCC 12.2. References to

⁶e.g., Linode or DigitalOcean VPSes - i.e., somebody else’s computer

⁷Perl iThreads, Coro, POE, MCE, AnyEvent, etc; though if one wished to see what Perl with threading built into the language would look and act like, take a look at the Qore Language, <https://docs.qore.org/current/lang/html/threading.html>.

⁸<https://metacpan.org/release/ODDLER/Alien-OpenMP-0.3.10>

⁹<https://github.com/Perl-OpenMP/>

later OpenMP functionality are limited to features that were already available in the tested GNU toolchain or are clearly identified as outside the experiments reported here.

Distribution	Version	GCC Version(s)
Ubuntu	7.10 (Gutsy Gibbon)	GCC 4.2
Fedora	7	GCC 4.1 (default), 4.2 (optional)
Fedora	8	GCC 4.2 (default)
Debian	5 (Lenny)	GCC 4.3 (default), GCC 4.2 (optional)
openSUSE	10.3	GCC 4.2
CentOS	5.x	GCC 4.1 (default), GCC 4.2 via manual installation
RHEL	5.x	GCC 4.1 (default), GCC 4.2 via Developer Toolset
RHEL	6.x	GCC 4.4 (default), GCC 4.8 via Developer Toolset

Table 1: Examples of Linux distributions and GCC versions relevant to the availability of GNU OpenMP support.

GCC	Year	OpenMP	C Support	C++	GFortran
4.2	2007	OpenMP 2.5	Basic (GOMP)	Basic (GOMP)	Basic
4.4	2009	OpenMP 3.0	Enhanced	Enhanced (reductions)	Improved
4.9	2014	OpenMP 4.0	Full	Full (C++11, tasking)	Full
5.1	2015	OpenMP 4.0	Optimizations	Updates (C++14)	Enhanced
6.1	2016	OpenMP 4.5	Optimizations	Updates (C++14)	Atomic operations
7.1	2017	OpenMP 4.5	More optimizations	Full	Improved tasking
8.1	2018	OpenMP 4.5	Full	Full	Full
9.1	2019	OpenMP 5.0 (partial)	Partial	Partial	Partial
10.1	2020	OpenMP 5.0 (expanded)	Expanded	Expanded	Expanded
11.1	2021	OpenMP 5.0/5.1 features	Ongoing	Ongoing	Ongoing
12.1	2022	OpenMP 5.0/5.1 features	Extensive	Extensive	Extensive
13.1	2023	OpenMP 5.x features	Ongoing	Ongoing	Ongoing
14.1	2024	OpenMP 5.x features	Ongoing	Ongoing	Ongoing

Table 2: Selected GCC OpenMP implementation milestones by version and language. For OpenMP 5.x, the specification listed denotes implemented features from that specification rather than complete conformance; see the GNU GOMP implementation-status documentation[2].

C Compiler	Year OpenMP Support Introduced	Languages Supported
Sun Studio (Oracle)	2005 (OpenMP 2.5)	C, C++, Fortran
GCC (GNU Compiler Collection)	2007 (OpenMP 2.5)	C, C++, Fortran
Intel C++ Compiler	2007 (OpenMP 2.5)	C, C++, Fortran
PGI (NVIDIA) Compiler	2007 (OpenMP 2.5)	C, C++, Fortran
IBM XL C/C++	2011 (OpenMP 3.1)	C, C++
IBM XL Fortran	2011 (OpenMP 3.1)	Fortran
GCC (GNU Compiler Collection)	2011 (OpenMP 3.1)	C, C++, Fortran
Intel C++ Compiler	2011 (OpenMP 3.1)	C, C++, Fortran
Clang (LLVM)	2013 (OpenMP 3.1)	C, C++, Fortran
Microsoft Visual C++	2012 (OpenMP 2.0)	C, C++

Table 3: Representative native compilers and selected OpenMP support milestones. The experiments reported in this work use Perl 5.40.0 built with GCC 12.2.0 and GNU `libgomp`. The current `Alien::OpenMP` project has subsequently added portability and CI coverage for additional compiler and platform combinations; those later improvements are not part of the experimental environment characterized here.

OpenMP presents an exceptionally rich opportunity for enhancing Perl on modern multi-core systems through `Inline::C`, given that it is available for use in Perl-related tasks literally *anywhere* a `perl` binary exists. The broader popularity of OpenMP continues to increase, and it will become even more ubiquitous as CPU core counts rise and GPUs find applications in extremely popular areas such as Artificial Intelligence ¹⁰. Its benefits for Perl extend far

¹⁰See Tim Mattson’s 2023 talk entitled, "How Popular is OpenMP?" - https://www.youtube.com/watch?v=Vk0ou-_hhTU

beyond Perl scripts, touching at the very core of the language. OpenMP was designed from the start for easily, incrementally, precisely, and conservatively parallelizing code in shared memory environments. It truly is a great complement to Perl on many levels.

3 Current Status of OpenMP Support in Perl

There are many execution models for scaling workloads with Perl. A common model is so-called *async*, or *fire and forget*. Two execution models are especially important here: the first is the traditional execution model that is enabled using Perl's `fork` keyword. The second kind is the one enabled by using OpenMP and `Inline::C`. In the former, the `perl` process obtains a child process with a logically duplicated address space when the `fork` keyword is encountered, often inside of some kind of a loop. In the latter, `Inline::C` is used to create a compiled shared object and create an XS interface so that it can be called inside of the Perl script like a regular Perl subroutine; except, the code that is executed is object code that runs outside the purview of the `perl` run-time. It is in this object code that OpenMP is used to enable real shared memory threads. The following sections delve a bit more into the details of each of these execution models.

3.1 `perl` + `fork`

Because the `perl` interpreter is a sequential process, fundamentally so¹¹, *basic* parallel execution in `perl` has been facilitated by POSIX `fork` as illustrated in Figure 1. A successful `fork` creates a child process with a logically duplicated virtual address space and initially equivalent process state. On Linux and other modern Unix-like systems this is normally implemented using copy-on-write pages, so the entire physical memory of the parent is not copied at the instant of the `fork`; pages can remain shared until either process writes to them, at which point affected pages are copied[4]. Thus, duplicating most of the parent's memory is better understood as a possible worst case for write-heavy children rather than an immediate cost of every `fork`. The parent and child nevertheless proceed as independent processes with separate address spaces for subsequent changes. Typically, the parent `perl` proceeds to some other work or generates additional child processes via `fork`, while each child continues to do something else—usually a specific task—until it has completed its job and `exits`. Multiple children may be spawned in this way, and there are many situations where this is useful, particularly when each child process is doing work that encapsulates high latency such as heavy disk I/O, network access, or interactions with a database server. `Parallel::ForkManager` is a useful module on CPAN for managing the spawning and reaping of completed `perl` child processes. The main compromise is that processes do not implicitly share subsequent Perl data updates; coordination requires inter-process communication, explicit shared memory, or some other middleware solution.

¹¹Its memory model and basis for maintaining a consistent interpreter state that is also highly optimized, depends deeply on the assumption of sequential memory access without any possibility of race conditions.

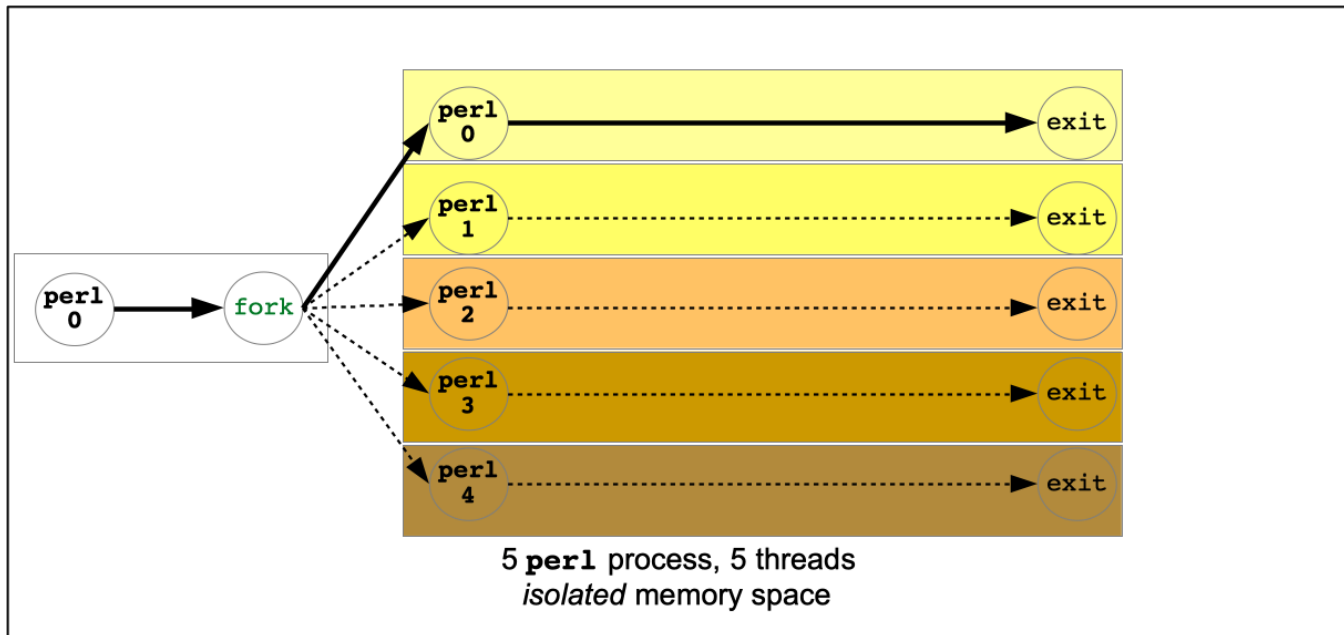


Figure 1: perl + fork: Conceptual process-parallel execution on a typical modern Unix-like system. A child begins with a copy-on-write view of the parent’s virtual address space, while parent and child remain distinct OS processes whose subsequent memory changes are isolated unless explicit inter-process communication or shared-memory mechanisms are used.

3.2 perl + OpenMP

The execution model used in this paper is illustrated in Figure 2. A single `perl` process calls a C function in a loaded shared object, such as one produced by `Inline::C`. When that native function enters an OpenMP `parallel` region, the OpenMP run-time creates worker threads inside the same process and shared address space. This is what the paper means by *running OpenMP inside Perl*. Useful work in that region often requires access to data that originated in Perl, so the central question becomes which Perl C API operations can be used safely by those native threads. The experiments deliberately begin with the most conservative case: reading established Perl data without invoking Perl callbacks or intentionally modifying interpreter-managed structures; more ambitious embedding patterns are discussed in [5].

Using OpenMP inside of a Perl script does not preclude the use of `fork`, but the ordering matters. A hybrid execution model can combine multiple `perl` processes with OpenMP threads inside each process, and copy-on-write can make such a design memory-efficient when large inherited data sets remain predominantly read-only. However, calling `fork` after an OpenMP run-time has already created worker threads requires care: the child process retains only the thread that called `fork`, while run-time state inherited from a previously multithreaded process can be inconsistent. This interaction has been a practical concern for GNU `libgomp` and is relevant whenever Perl code mixes process parallelism with OpenMP-enabled native libraries[6].

OpenMP 5.0 introduced `omp_pause_resource` and `omp_pause_resource_all`, which allow an OpenMP run-time to relinquish resources and provide one mechanism for preparing a process before a subsequent `fork`[7]. These routines were already implemented by GNU `libgomp` beginning with GCC 9, so they were available in the GCC 12.2.0 environment used for the experiments in this paper[2]. The OpenMP 5.0 API requires these calls to occur outside an explicit parallel region, with explicit tasks completed before the run-time is paused[7]. They were not themselves exercised by the tests reported here. The conservative pattern remains to create child processes before initializing OpenMP where practical, or to deliberately quiesce the OpenMP run-time before `fork`. Other OpenMP run-times may behave differently, but they were outside the tested compiler scope of this work. A complete treatment of hybrid process/thread execution, including oversubscription and interactions with independently threaded libraries, remains outside the scope of this paper.

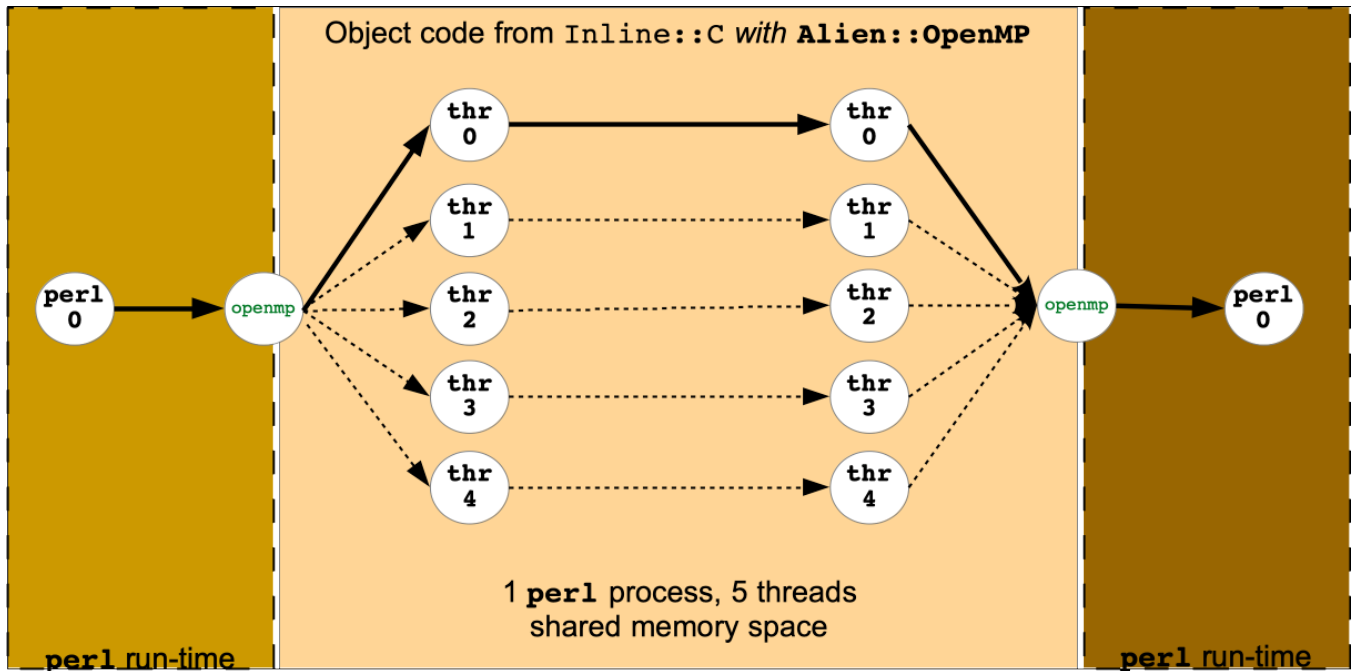


Figure 2: perl + OpenMP execution model used in this work: a single perl process calls compiled C code through `Inline::C`, and the OpenMP run-time creates worker threads within that process. Other native interfaces, including hand-written XS or an FFI-loaded library, can introduce the same shared-address-space OpenMP execution model.

3.3 `OpenMP::Environment`

The process environment plays an important role in OpenMP run-time configuration. `OpenMP::Environment` does not alter C code or OpenMP compilation; it provides Perl accessors for environment variables that influence a running OpenMP program. These accessors set or query values through Perl’s global `%ENV` hash. Figure 6 shows `OMP_NUM_THREADS` being updated from Perl before an OpenMP-enabled C function is called. Section 4 discusses why this matters when the OpenMP run-time is already loaded into a long-lived perl process.

The module has continued to develop since the experiments reported here. The current CPAN release, `OpenMP::Environment` 1.5.0, adds support for additional OpenMP 5.2/libgomp environment controls, lvalue accessors, opt-in `assert/unset` DSL shortcuts, and a separate validation policy module with portable grammar and cross-variable checks.¹² Those later interfaces improve the present-day project but are not assumed by the experimental code or results in this paper.

3.4 `Inline::C`

The approach used throughout this paper to include OpenMP in Perl scripts relies fundamentally on `Inline::C`. In particular, `Alien::OpenMP` and `OpenMP::Simple` work directly as plug-ins with¹³ `Inline::C`. It provides a convenient way to create C-backed subroutines directly alongside Perl code by parsing the C section and generating the XS and build machinery needed to compile, link, and load it into the running Perl program. OpenMP can certainly be used from hand-written XS by including `#include <omp.h>` and supplying the appropriate compiler and linker flags; `Inline::C` is used here because it keeps those interface details compact and makes the examples more accessible to the intended audience. Figure 3 is a snippet of the XS code generated by `Inline::C` for the test written to investigate the Perl API macros and functions in Table 5.

¹²<https://metacpan.org/release/OODLER/OpenMP-Environment-1.5.0>

¹³See <https://metacpan.org/dist/Inline-C/view/lib/Inline/C/Cookbook.pod#ENTERTAINING-GUESTS>.

```

1  #include "EXTERN.h"
2  #include "perl.h"
3  #include "XSUB.h"
4  #include "INLINE.h"
5  #include <omp.h>
6  #include "/home/user/perl5/perlbrew/perl5/perl-5.40.0/lib/site_perl/5.40.0/auto/share/dist/OpenMP-Simple/ppport.h"
7  #include
   ↪  "/home/user/perl5/perlbrew/perl5/perl-5.40.0/lib/site_perl/5.40.0/auto/share/dist/OpenMP-Simple/openmp-simple.h"
8
9  /* C function parallelized with OpenMP */
10 int _check_num_threads() {
11     int ret = 0;
12
13     PerlOMP_GETENV_BASIC
14
15     #pragma omp parallel
16     {
17         #pragma omp single
18         ret = omp_get_num_threads();
19     }
20
21     return ret;
22 }
23
24 // Remaining generated C interfaces are omitted from this excerpt.
25
26 MODULE = _01_scalar_t_f17b  PACKAGE = main
27
28 PROTOTYPES: DISABLE
29
30
31 int
32 _check_num_threads ()
33
34 SV *
35 testSvPV (input)
36     SV *    input
37
38 SV *
39 testSvIV (input)
40     SV *    input
41
42 SV *
43 testSvNV (input)
44     SV *    input
45
46 SV *
47 testSvTRUE (input)
48     SV *    input

```

Figure 3: Excerpt from the XS wrapper generated by `Inline::C` for the scalar (SV) test harness, including the OpenMP and `OpenMP::Simple` headers and generated Perl-callable interfaces.

3.5 `Alien::OpenMP`

For the purposes of this work, `Inline::C` provides the shortest path from a Perl program to OpenMP-enabled C while still exposing the generated XS and Perl C API when deeper inspection is useful. This keeps the focus on the Perl/OpenMP boundary rather than on the mechanics of hand-maintaining XS build scaffolding.

This module provides the configuration that `Inline::C` needs to compile and link OpenMP-enabled C code. For GCC, the essential compiler and linker flag is `-fopenmp`. `Alien::OpenMP` can be used directly, as shown in Figure 4. The higher-level `OpenMP::Simple` module builds on `Alien::OpenMP` and is discussed in Section 3.6.

The current `Alien::OpenMP` release is 0.3.10, with broader compiler-selection and CI coverage than the GCC-only environment used to produce the results in this paper. Current CI includes Linux GCC and Clang, Strawberry Perl on Windows, Apple Clang on macOS, FreeBSD, and GCC 10–16 builds. This later portability work is important to the practical reach of the `Perl+OpenMP` Project, but no result in the following experimental sections should be read as having been reproduced on those newer compilers, platforms, or Perl versions.

```

1  #!/usr/bin/env perl
2  use strict;
3  use warnings;
4  use Alien::OpenMP;
5  use Inline (
6      C          => 'DATA',
7      with       => qw/Alien::OpenMP/,
8  );
9
10 # Remaining Perl code is omitted from this excerpt.
11
12 __DATA__
13 __C__
14
15 // Remaining C code is omitted from this excerpt.

```

Figure 4: Using `Alien::OpenMP` as an `Inline::C` plug-in. The `with => qw/Alien::OpenMP/` option supplies the compiler and linker configuration needed to build the inlined C code with the GNU OpenMP run-time.

3.6 `OpenMP::Simple`

This module goes a step beyond `Alien::OpenMP` by providing run-time C helpers and macros for recurring OpenMP/Perl integration needs. It began with the need to reread environment settings such as `OMP_NUM_THREADS` before successive calls into an already loaded OpenMP-enabled shared library. Figure 5 shows this basic use, and Table 4 summarizes the run-time setter routines and `OpenMP::Simple` helpers relevant to the approach used in this work. The module has since grown to support array conversion and validation helpers as well. Arrays remain a natural priority because OpenMP commonly operates on homogeneous data such as `int` and `float`, while Perl also creates practical demand for strings and other data representations.

The current CPAN release, `OpenMP::Simple` 0.2.7, substantially expands CI and portability testing and also tightens one important Perl/native boundary. Its threaded string-conversion helper now performs Perl API access and allocation on the calling Perl thread, then allows OpenMP workers to copy only between native buffers.¹⁴ That change is a useful example of the conservative principle developed in this paper: where Perl interpreter ownership is uncertain or platform-sensitive, stage the Perl-facing work before entering the worker-thread region and parallelize the native operation beyond that boundary. Other conversion helpers remain useful subjects for the exact access-pattern analysis advocated here.

¹⁴<https://metacpan.org/release/OODLER/OpenMP-Simple-0.2.7>

Name	Description
<code>omp_set_num_threads</code>	Sets the upper team size limit for parallel regions.
<code>omp_set_schedule</code>	Defines the run-time scheduling method for loops in parallel regions.
<code>omp_set_dynamic</code>	Enables or disables dynamic adjustment of team sizes during execution.
<code>omp_set_nested</code>	Controls whether nested parallel regions are allowed or not.
<code>omp_set_max_active_levels</code>	Limits the number of active parallel regions that can run concurrently.
<code>omp_set_default_device</code>	Sets the default device for target regions, specifying whether computations should run on CPU or GPU.
<code>omp_set_num_teams</code>	Sets the upper limit on the number of teams for the teams construct in OpenMP.
<code>omp_set_teams_thread_limit</code>	Defines the maximum number of threads per team in the teams construct.
<code>PerlOMP_UPDATE_WITH_ENV__DEFAULT_DEVICE</code>	Updates the default device setting from the environment variable <code>OMP_DEFAULT_DEVICE</code> .
<code>PerlOMP_UPDATE_WITH_ENV__TEAMS_THREAD_LIMIT</code>	Updates the thread limit for teams from the environment variable <code>OMP_TEAMS_THREAD_LIMIT</code> .
<code>PerlOMP_UPDATE_WITH_ENV__NUM_TEAMS</code>	Updates the number of teams for OpenMP from the environment variable <code>OMP_NUM_TEAMS</code> .
<code>PerlOMP_UPDATE_WITH_ENV__MAX_ACTIVE_LEVELS</code>	Updates the maximum active levels from the environment variable <code>OMP_MAX_ACTIVE_LEVELS</code> .
<code>PerlOMP_UPDATE_WITH_ENV__NUM_THREADS</code>	Updates the number of threads for OpenMP from the environment variable <code>OMP_NUM_THREADS</code> .
<code>PerlOMP_UPDATE_WITH_ENV__DYNAMIC</code>	Enables or disables dynamic teams from the environment variable <code>OMP_DYNAMIC</code> .
<code>PerlOMP_UPDATE_WITH_ENV__NESTED</code>	Enables or disables nested parallelism from the environment variable <code>OMP_NESTED</code> .
<code>PerlOMP_UPDATE_WITH_ENV__SCHEDULE</code>	Updates the scheduling method and chunk size from the environment variable <code>OMP_SCHEDULE</code> .
<code>PerlOMP_GETENV_BASIC</code>	A bundled macro that updates environment variables for <code>OMP_NUM_THREADS</code> and <code>OMP_SCHEDULE</code> .
<code>PerlOMP_RET_ARRAY_REF_ret</code>	Initializes a new Perl array reference and makes it mortal for proper garbage collection.
<code>PerlOMP_1D_Array_NUM_ELEMENTS</code>	Returns the number of elements in a 1D Perl array reference.
<code>PerlOMP_1D_Array_TO_1D_FLOAT_ARRAY</code>	Converts a 1D Perl array reference into a C array of floats.
<code>PerlOMP_1D_Array_TO_1D_FLOAT_ARRAY_r</code>	A threaded version of the function that converts a 1D Perl array reference into a C array of floats.
<code>PerlOMP_1D_Array_TO_1D_INT_ARRAY</code>	Converts a 1D Perl array reference into a C array of integers.
<code>PerlOMP_1D_Array_TO_1D_INT_ARRAY_r</code>	A threaded version of the function that converts a 1D Perl array reference into a C array of integers.
<code>PerlOMP_2D_AoA_TO_2D_FLOAT_ARRAY</code>	Converts a 2D Perl array of arrays (AV of AVs) into a 2D C array of floats.
<code>PerlOMP_2D_AoA_TO_2D_FLOAT_ARRAY_r</code>	A threaded version of the function that converts a 2D Perl array of arrays (AV of AVs) into a 2D C array of floats.
<code>PerlOMP_2D_AoA_TO_2D_INT_ARRAY</code>	Converts a 2D Perl array of arrays (AV of AVs) into a 2D C array of integers.
<code>PerlOMP_2D_AoA_TO_2D_INT_ARRAY_r</code>	A threaded version of the function that converts a 2D Perl array of arrays (AV of AVs) into a 2D C array of integers.

Table 4: OpenMP run-time setter routines used by `OpenMP::Simple` together with `PerlOMP_*` environment-update, conversion, and helper routines defined in `openmp-simple.h`, as used in this work.

```

1 use strict;
2 use warnings;
3 use OpenMP::Simple;
4 use OpenMP::Environment;
5 use Inline (
6     C => 'DATA',
7     with => qw/OpenMP::Simple/,
8 );
9 my $env = OpenMP::Environment->new;
10 for my $want_num_threads ( 1 .. 8 ) {
11     $env->omp_num_threads($want_num_threads);
12     $env->assert_omp_environment;
13     # call parallelized C function
14     my $got_num_threads = _check_num_threads();
15     printf "%0d threads spawned in ".
16           "the OpenMP run-time, expecting %0d\n",
17           $got_num_threads, $want_num_threads;
18 }
19 __DATA__
20 __C__
21 int _check_num_threads() {
22     PerlOMP_GETENV_BASIC
23     int ret = 0;
24     #pragma omp parallel
25     {
26         #pragma omp single
27         ret = omp_get_num_threads();
28     }
29     return ret;
30 }

```

Figure 5: Example of `OpenMP::Environment` managing `OMP_NUM_THREADS` through Perl’s `%ENV` hash while `OpenMP::Simple`’s `PerlOMP_GETENV_BASIC` macro applies selected environment settings to the already loaded OpenMP run-time before entering the parallel region.

`Inline::C` is not the only way an OpenMP run-time can enter a Perl process. A foreign-function interface such as `FFI::Platypus` can call an existing native library exposed through a C-compatible ABI, including libraries implemented in C, C++, or Fortran, that was itself compiled with OpenMP, as discussed in [6]. In that case the Perl program may not directly contain any OpenMP directives at all, yet the loaded library can still initialize an OpenMP run-time and create worker threads. The experiments in this paper did not exercise an FFI path, so no claim about FFI-specific behavior is made here; the important point is that the hybrid `fork`+OpenMP considerations in Section 3.2 apply equally when OpenMP is introduced indirectly through a native library. This also suggests a useful future direction for `OpenMP::Simple`: providing Perl-facing helpers around the OpenMP 5.0 resource-pause routines when a program must deliberately quiesce a GNU OpenMP run-time before process creation.

3.7 `OpenMP.pm`

This is a *meta* module that pulls together the principal `Perl+OpenMP` Project components. Figure 6 illustrates the use of its constructor (`OpenMP->new`) that returns an object reference providing an accessor to the `OpenMP::Environment` instance, `$omp->env`. The expected output is shown in Figure 7. See more about `OpenMP::Environment` and `OpenMP::Simple` in Sections 3.3 and 3.6, respectively.

The current CPAN release, `OpenMP 1.0.3`, serves as the project installation point and declares dependencies on `Alien::OpenMP`, `Inline::C`, `OpenMP::Environment`, `OpenMP::Simple`, and `perlomp`. Thus a current installation also provides the general project manual through `perldoc perlomp`.¹⁵ The straightforward example in Figure 6 is adapted from the `Perl+OpenMP` Project project page on GitHub.

3.8 Current Project Integration and Remaining Plans

The general project manual that was an earlier future goal now exists as the CPAN distribution `perlomp 0.01`. It provides a Perl-style manual page for installation, project components, run-time initialization, environment validation, portability, native-thread safety considerations, and quick starts for `Inline::C` and external OpenMP executables.¹⁶ A separate cookbook-oriented treatment remains useful future work, potentially as `perldoc perlompcook`.

¹⁵<https://metacpan.org/release/OODLER/OpenMP-1.0.3>

¹⁶<https://metacpan.org/dist/perlomp>

Other ongoing goals include continued portability hardening, additional helpers for moving deliberately between Perl and native data representations, and patterns for hybrid process/thread applications. User feedback remains important for prioritizing interfaces that reduce the friction of combining Perl and OpenMP without obscuring the ownership and synchronization boundaries emphasized in this paper.

```

1  #!/usr/bin/env perl
2  use strict;
3  use warnings;
4  use OpenMP;
5  use Inline (
6      C => 'DATA',
7      with => qw/OpenMP::Simple/,
8  );
9
10 my $omp = OpenMP->new;
11
12 for my $want_num_threads ( 1 .. 8 ) {
13     $omp->env->omp_num_threads($want_num_threads);
14     $omp->env->assert_omp_environment;
15
16     # call parallelized C function
17     my $got_num_threads = _check_num_threads();
18     printf "%0d threads spawned in ".
19         "the OpenMP run-time, expecting %0d\n",
20         $got_num_threads, $want_num_threads;
21 }
22
23 __DATA__
24 __C__
25
26 /* C function parallelized with OpenMP */
27 int _check_num_threads() {
28     int ret = 0;
29     PerlOMP_GETENV_BASIC // reads OMP_NUM_THREADS, updates run-time
30     #pragma omp parallel
31     {
32         #pragma omp single
33         ret = omp_get_num_threads();
34     }
35     return ret;
36 }
37 __END__
38

```

Figure 6: “Quick Start” example from the Perl+OpenMP Project project page, showing Perl-side OpenMP environment configuration and invocation of an OpenMP-enabled Inline::C function.

```

1  1 threads spawned in the OpenMP run-time, expecting 1
2  2 threads spawned in the OpenMP run-time, expecting 2
3  3 threads spawned in the OpenMP run-time, expecting 3
4  4 threads spawned in the OpenMP run-time, expecting 4
5  5 threads spawned in the OpenMP run-time, expecting 5
6  6 threads spawned in the OpenMP run-time, expecting 6
7  7 threads spawned in the OpenMP run-time, expecting 7
8  8 threads spawned in the OpenMP run-time, expecting 8

```

Figure 7: Expected output from the “Quick Start” example in Figure 6, showing the requested OpenMP thread count applied on successive calls.

4 The Critical Role of %ENV

The \$USER’s environment plays a crucial role in shaping user expectations and assumptions in both phases related to OpenMP: *compile-time* processing and during program *execution*. The user could be a developer or someone running the parallel program - often, they are one and the same. The Perl+OpenMP Project aims to replicate how

OpenMP treats the environment as accurately as possible, respecting the intentional role the `$USER` environment plays in the OpenMP specifications. To this end, the `OpenMP::Environment` module was created as the very first in a series of related modules. In fact, the creation of this module preceded support for OpenMP *with Inline::C* via the `Alien::OpenMP` module, which is discussed in Section 3.5.

To fully appreciate the importance of `%ENV` to OpenMP, one must understand the role of the `$USER` environment in the OpenMP run-time. Traditionally, OpenMP programs consist of single language that are compiled into standalone executable binaries. When executed, users can control the number of threads spawned in the first `#pragma omp parallel` section by setting the `OMP_NUM_THREADS` environment variable. At run-time, the OpenMP run-time checks this variable. If set, this value defines the number of threads created. If the variable is not set, a default value is used. The programmer can use the standard OpenMP run-time function `omp_set_num_threads(int num_threads)` to update this after the run-time starts.

In the case of standalone executables run from the command line, the value of `OMP_NUM_THREADS` is used only at the start. To adjust the number of threads during execution, the program must call `omp_set_num_threads` directly, as the OpenMP run-time doesn't re-read `OMP_NUM_THREADS`. It is common practice to set this environment variable for the program's duration, assuming no changes are needed once execution begins. For more complex use cases, such as running benchmarks, a Unix shell script might be used to iterate over different values for `OMP_NUM_THREADS`, updating the variable and rerunning the executable.

```

1  #!/usr/bin/bash
2  for i in 1 2 4 8 16 32 64; do
3      OMP_NUM_THREADS=$i ./omp-program.x # set for this launch; read by the OpenMP run-time
4  done

```

Figure 8: Conceptual shell-side pattern for varying `OMP_NUM_THREADS` between launches of a standalone OpenMP executable (here, `omp-program.x`), so that each newly started process can initialize its OpenMP run-time from the process environment.

In the context of running OpenMP code inside of a Perl script and given the dynamic nature of most Perl scripts, the value of `OpenMP::Environment` should now start to be clear. The idea is to enable the usage pattern seen in Figure 8. But there was a small problem that had to be overcome before this was possible. `Inline::C` works by parsing the C program that is *inlined* into the Perl program. It then compiles that C code into *temporary* shared object libraries; then it generates the necessary XS code needed to import these shared libraries into the `perl` process that is interpreting the Perl script, with the expected interface that the Perl programmer would use to call this function as if it were any other Perl subroutine. The key difference here is that, instead of being a standalone executable, the OpenMP-enabled C code is now part of a shared library that `perl` loads exactly *once*. And whereas the Unix shell script is creating a new process on each iteration and therefore a new OpenMP run-time¹⁷, the Perl script is running in the same process the whole time. Therefore, there is no automatic reload point at which the OpenMP run-time rereads the latest value of `OMP_NUM_THREADS`.

The code in Figure 6 shows exactly this situation. It also includes the solution, which is the addition of the C macro, `PerlOMP_GETENV_BASIC` on Line #29. This macro actually uses 2 more macros; one reads `OMP_NUM_THREADS` and the other reads and sets `OMP_SCHEDULE`. Figure 9 shows the core C logic used by the macro. At run time it reads `OMP_NUM_THREADS` using C's `getenv` function and applies the value through the standard OpenMP run-time routine `omp_set_num_threads` before the next parallel region. This macro and the other macros provided by `OpenMP::Simple` are there to make it as easy as possible to get started writing Perl programs that contain C parallelized using OpenMP. The most current implementation of these macros may be seen on `OpenMP::Simple`'s GitHub¹⁸.

```

1  char *num5 = getenv("OMP_NUM_THREADS");
2  if (num5 != NULL) {
3      omp_set_num_threads(atoi(num5));
4  }

```

Figure 9: Core C logic used by the `OpenMP::Simple` helper for rereading `OMP_NUM_THREADS`: the value is obtained with `getenv()` and applied to the active OpenMP run-time with `omp_set_num_threads()`.

¹⁷that also reads the `$USER`'s environment to get the new value for `OMP_NUM_THREADS`)

¹⁸<https://github.com/Perl-OpenMP/p5-OpenMP-Simple/blob/master/share/openmp-simple.h>

5 The Effectiveness of Perl with OpenMP

The remainder of this paper explores what happens when Perl and OpenMP are combined. The interface is indirect: OpenMP operates in compiled native code reached through Perl's extension mechanisms. In the experiments reported here, `Inline::C` provides that bridge. It removes much of the build and XS boilerplate, but the programmer must still design subroutine interfaces that are safe, idiomatic, and understandable to Perl callers. When Perl programmers first discover `Inline::C`, they naturally tend to focus on the C implementation, and the resulting subroutine signatures can therefore look more C-oriented than idiomatic Perl. As familiarity with Perl's C API grows, it becomes natural to accept Perl references directly, including references to arrays and hashes. OpenMP-oriented C code also frequently operates on multidimensional arrays, making this interface boundary especially important.

5.1 The perl Process and OpenMP Threads

A normal `perl` invocation begins as a single operating-system process. Comparing Figure 1 with Figure 2 highlights the distinction between creating additional processes with `fork` and creating OpenMP worker threads within one process. When a program is executing a parallel OpenMP section as in Figure 2, the initial thread of the `perl` process becomes the OpenMP **master** thread (i.e., the OpenMP thread with `id` of 0). With OpenMP, the process is treated as a single thread until it encounters a parallel section, e.g., `#pragma omp parallel`. From the programmer's point of view, when this directive is encountered, the OpenMP run-time spawns a team whose size is governed by `OMP_NUM_THREADS` and the applicable run-time settings (unless this has been adjusted during run time, for example by calling the `omp_set_num_threads` run-time function). Once that happens the threads run according to the other OpenMP directives that are present inside of the parallel region. As far as the operating system is concerned, this is all happening under a single logical process. All the threads that have been created and are now running have access to the entire set of the original process' memory. In this case, the process started as a single `perl` process with a single logical thread, when execution is inside of the OpenMP parallel section, it is still a single `perl` process, but multiple native threads are now executing code within that process. This should be very exciting for any Perl programmer who has wanted *real* shared memory threads to be available to `perl`. It's not ideal, for example the threads are not inside the Perl run-time, *per se*. The threads are inside of executing object code that has been injected into a `perl` process and given a subroutine interface via `Inline::C`. So taking advantage of threading in this environment is going to involve using C and the Perl API - not exactly the high level Perl experience most would prefer.

```

extern _INT32 main() {
    register _INT32 _w2c__ompv_ok_to_fork;
    register _UINT64 _w2c__reg3;
    register _INT32 _w2c__comma;
    _INT32 my_id;
    _INT32 __ompv_gtid_s1;

    /*Begin_of_nested_PU(s)*/

    _w2c__ompv_ok_to_fork = 1;
    if(_w2c__ompv_ok_to_fork)
    {
        _w2c__ompv_ok_to_fork = __ompc_can_fork();
    }
    if(_w2c__ompv_ok_to_fork)
    {
        __ompc_fork(0, &__omprg_main_1, _w2c__reg3);
    }
    else
    {
        __ompv_gtid_s1 = __ompc_get_local_thread_num();
        __ompc_serialized_parallel();
        _w2c__comma = omp_get_thread_num();
        my_id = _w2c__comma;
        printf("hello from %d\n", my_id);
        __ompc_end_serialized_parallel();
    }
    return 0;
} /* main */

```

calls RTL fork and passes function pointer to outlined main()

__omprg_main_1's frame pointer

serial version

Figure 10: Example of lowered C code generated by OpenUH from source containing OpenMP directives, illustrating the lower-level run-time and threading machinery that an OpenMP compiler can generate on the programmer's behalf[8].

5.2 Importance of Accepting and Returning Perl References (SV)

For any interface between Perl and C, particularly in the context of OpenMP parallelization, it is crucial that the subroutine interface in `Inline::C` can seamlessly handle Perl references, such as `AV` (Perl arrays), `HV` (Perl hashes), and return the same kind of references via `SV` (scalar references). This is vital because Perl's flexibility relies heavily on references to build complex and dynamic data structures. Without the ability to accept and return these references, integrating Perl with OpenMP would significantly complicate matters, especially in terms of presenting idiomatic subroutine signatures and parameter unpacking in the C functions.

The opportunity to use Perl references directly in C functions allows for a cleaner, more natural interaction between C and pure Perl. This can respect Perl's internal data management and avoid unnecessary conversion into simpler C structures when the Perl representation itself is suitable for the operation. Conversion is not always avoidable, however, and for numerical kernels it may be desirable to transform Perl data into a native representation. When conversion is necessary, it should be performed as infrequently as practical: validate and convert at the boundary of a computational kernel, do substantial work on the native representation, and convert results back only when needed. Repeated Perl-to-C coercion inside a hot parallel loop can erase much of the benefit that motivated the native kernel in the first place. By allowing functions to accept and return SV values, including array and hash references, developers can still keep Perl's memory management intact and present an idiomatic Perl interface while choosing deliberately where native buffers are warranted; related approaches are discussed in [6].

This approach is not uniquely enabled by OpenMP. Native threading interfaces such as POSIX threads, and where available the C language's native threading facilities, can also be used from `XS` or `Inline::C` code. A programmer could therefore build a custom thread pool, synchronization scheme, scheduler, reductions, and work distribution directly in C. The attraction of OpenMP is that these lower-level mechanisms are exposed through a mature higher-level programming model: parallel regions, loop work-sharing, reductions, scheduling, tasks, synchronization, and run-time controls can be expressed without reimplementing all of that machinery for each extension. OpenMP is thus not a replacement for native threads; it is an abstraction that is particularly useful when the work can be described naturally in OpenMP terms. Figure 10 illustrates the kind of lower-level code an OpenMP source-to-source compiler may generate from directives. `Alien::OpenMP` and `Inline::C` then address a different part of the problem by supplying the compiler and linker configuration needed to make that OpenMP-enabled native code callable from Perl.

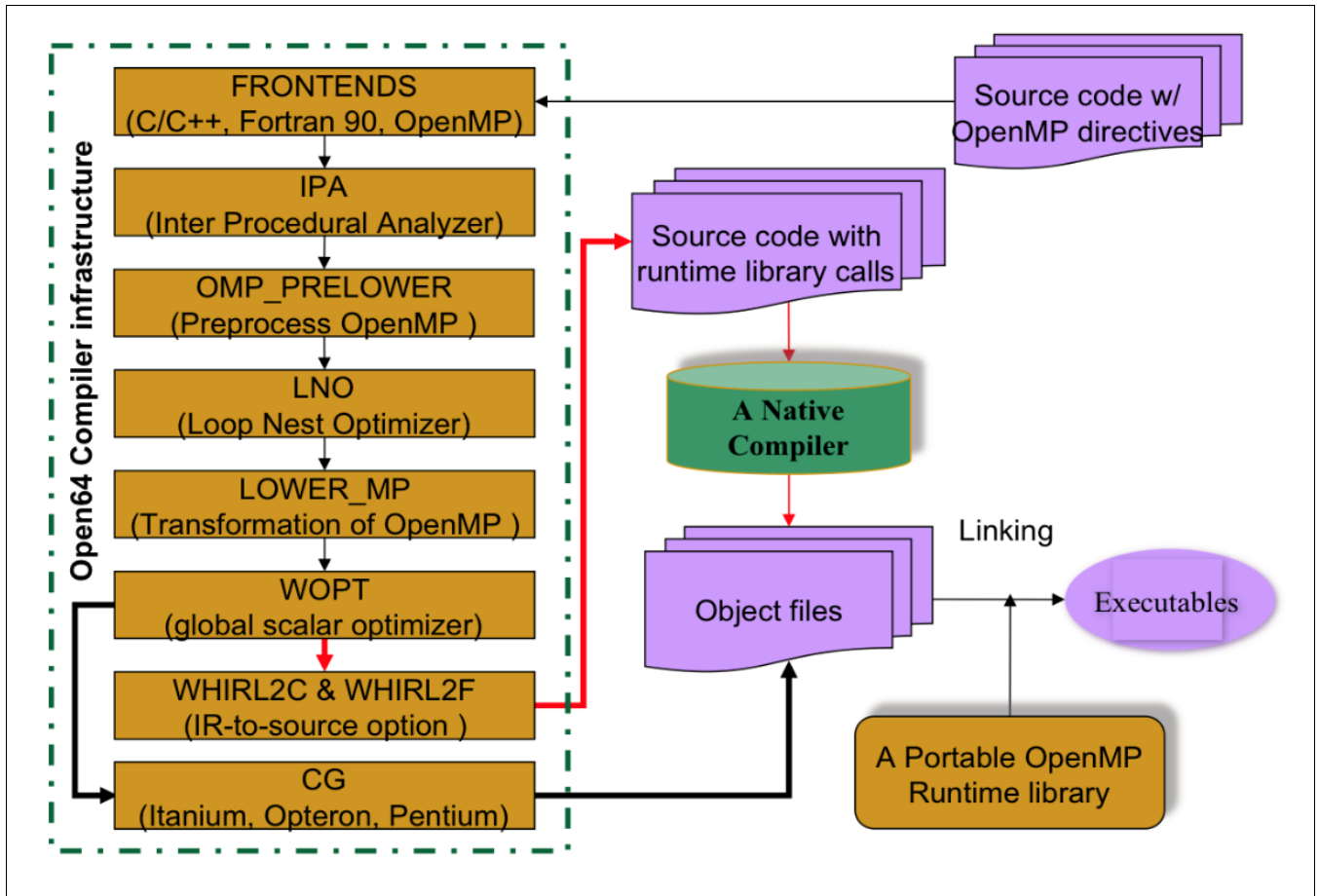


Figure 11: OpenUH compilation pipeline for transforming source containing OpenMP directives into lowered code and run-time calls, based on the optimizing portable OpenMP compiler described by Liao et al.[8].

The importance of getting this interface correct, becomes particularly evident when OpenMP is involved. Parallelized Perl subroutines must interact with complex Perl data structures safely and idiomatically. Ensuring that `Inline::C` functions can easily interface with AV and HV references is essential for making the integration between Perl and OpenMP both seamless and effective. This setup not only increases the usability of OpenMP in Perl scripts but also ensures that such scripts remain maintainable and flexible in the long run.

5.3 Dealing with Perl Arrays in OpenMP Programs

When working with OpenMP, much parallelization involves arrays, especially arrays of numbers of the same native C type, including 2D or 3D arrays. OpenMP data-parallel loops are particularly natural over contiguous native buffers. A Perl AV must not, however, be confused with a native C numerical array. The AV contains an internal contiguous C array of SV * pointers; the SV objects reached through those pointers, and the payloads represented by those SV objects, are not thereby laid out as one contiguous `int[]` or `double[]` buffer[9]. This distinction is important both for correctness and for reasoning about cache locality and conversion costs.

Therefore, there are two useful Perl/OpenMP idioms rather than a single rule that Perl data must always be converted. Read-only operations that can safely use the Perl C API may work directly with an AV and its elements, which is a major subject of this paper. Computational kernels that benefit from a homogeneous native representation may instead convert the Perl values once into a contiguous C buffer, perform the OpenMP work there, and convert results back only as necessary. This boundary-oriented approach avoids implying that a Perl array is physically equivalent to a C numerical array while preserving the practical advantages of each representation. Understanding where that boundary should be placed is essential when passing data between `Inline::C` and Perl. For sufficiently sophisticated numerical array work, the *Perl Data Language*¹⁹ may also be worth considering.

¹⁹<https://pdl.perl.org>

5.4 Potential Uses of HV in OpenMP Subroutines

In the context of OpenMP and Perl, using hash references (HV) in parallelized subroutines opens up exciting opportunities for handling and processing complex, non-linear data structures. Hashes in Perl offer an elegant way to represent key-value pairs, making them ideal for organizing data in sophisticated manners, such as look-up tables, non-relational data (similar to SQL), or even graph structures. These data types are particularly valuable in computational problems that require mapping, querying, and manipulating data in a structured but flexible manner. For example, in many scientific and engineering applications, look-up tables (LUTs) are commonly used to map input values to corresponding outputs. In Perl, this can be efficiently modeled using HVs, which allow for fast lookups and updates. Similarly, non-relational data structures that don't conform to rigid SQL schemas can be modeled by HVs, enabling dynamic and flexible data handling. A key advantage of this approach is the ability to manage heterogeneous data types within a single structure, offering a more robust solution compared to other data representations.

In graph-based problems, HVs can represent nodes and edges with keys and values, respectively, which is useful for algorithms like shortest path, network flow, or even complex data-driven simulations. This allows for parallelization of graph traversal or manipulation tasks with OpenMP, enhancing performance by distributing computational load across multiple threads. While this paper doesn't delve into OpenMP tasking specifically, it is worth noting that tasking offers an alternative approach to parallel processing data, especially for structures like HVs that may not fit neatly into a linear, predictable iteration pattern. Tasking allows work to be divided dynamically based on a *producer-consumer* paradigm that makes pools of work available to computational resources, rather than relying solely on `#pragma omp parallel for` loops for more tightly integrated work-sharing. Tasking became part of OpenMP 3.0 and is supported by GCC beginning with GCC 4.4, and has since become an integral feature of OpenMP, allowing for more flexibility in handling irregular data processing and improving overall parallel performance for certain workloads. By combining HVs and OpenMP, Perl developers can harness the power of parallelism while maintaining the expressiveness and flexibility of Perl's data structures. For more information on related topics, see Section 8.8.

6 Testing Multi-Threaded Reading of Perl Data Structures

A common refrain heard when discussing the Perl API and OpenMP is that the API is not thread-safe. As a general guarantee, that is correct: the public Perl C API was not designed to promise safe concurrent access from arbitrary native threads. The narrower question investigated here is whether particular access patterns can remain non-mutating and behave safely when multiple OpenMP threads read the same Perl data structure. The purpose of the tests is therefore empirical and conservative: identify useful operations that behaved safely under specified read-only conditions, document the conditions, and distinguish those observations from any general guarantee by the Perl API. Where an operation can mutate interpreter state, serialization or a redesign of the interface remains necessary.

6.1 Not All Perl API Reads Are Passive

Even an operation that looks like a read at the call site may have a mode that changes interpreter-managed state. The clearest example encountered here is `av_fetch`: its `lval` argument determines whether an out-of-range access may grow the AV. Table 7 summarizes that interface. Passing `lval = 0` requests a non-growing fetch, which is the form exercised by the concurrent-read tests in this paper. Hash iteration presents a different problem: `hv_iterinit` and `hv_iternext` maintain iterator state associated with the HV, so multiple worker threads cannot independently advance that same iterator without coordination. The direct bucket traversal explored later with `HvARRAY` and `HvMAX` avoids that shared iterator state, but only while the hash remains structurally unchanged. These examples are why “read-only” must describe the actual behavior of an access path, not merely the programmer's intention.

6.2 Investigation Methodology

For each of the three Perl data structure types investigated (SV, AV, and HV), a stress-test program²⁰ was written using the `OpenMP` module, which loads `OpenMP::Environment` and `OpenMP::Simple`. Candidate API functions and macros were selected because their intended use appeared compatible with read-only access. Each test repeatedly called OpenMP-enabled C functions while increasing the requested thread count, and `Test2` assertions checked the returned values. The goal was not to prove a universal thread-safety property, but to establish a small, reproducible set of access patterns that behaved correctly under the tested conditions without adding synchronization solely around the read itself.

²⁰<https://github.com/Perl-OpenMP/p5-OpenMP-Tests-Readonly>

6.3 Safe API Functions for SV (Scalars)

Function	Description	Observed Safe?
<code>SvPV()</code>	Fetches the pointer to the string value.	Yes
<code>SvIV()</code>	Retrieves the integer value of the scalar.	Yes
<code>SvNV()</code>	Retrieves the numeric (double) value.	Yes
<code>SvTRUE()</code>	Evaluates the truthiness of the scalar.	Yes
<code>SvTYPE()</code>	Returns the type of the scalar.	Yes
<code>SvCUR()</code>	Retrieves the length of the string data in the scalar.	Yes
<code>SvLEN()</code>	Fetches the allocated size of the scalar’s string buffer.	Yes
<code>SvREFCNT()</code>	Returns the reference count.	Yes

Table 5: Selected Perl scalar API access macros and their observed behavior under the concurrent read-only tests reported in this work (Perl 5.40.0, GCC 12.2.0). “Yes” records the outcome of these tests rather than a general Perl API thread-safety guarantee.

All of the Perl API access macros listed in Table 5 produced successful results under the concurrent read-only test conditions represented in Figure 12; the full source is on GitHub²¹. These results are encouraging because the tested operations inspect established scalar state without intentionally changing the scalar or its reference count. They should nevertheless be read as results for the values and access patterns exercised by the test suite, not as a blanket guarantee that every invocation of a similarly named Perl macro is side-effect free for every possible SV state.

²¹<https://github.com/Perl-OpenMP/p5-OpenMP-Tests-Readonly/blob/master/t/01-scalar.t>

```

1  #!/usr/bin/env perl
2  use strict;
3  use warnings;
4  use Test2::V0;
5  use OpenMP;
6  use Inline (
7      C => 'DATA',
8      with => qw/OpenMP::Simple/,
9  );
10 my $omp = OpenMP->new;
11 for my $want_num_threads ( 1 .. 16 ) {
12     $omp->env->omp_num_threads($want_num_threads);
13     $omp->env->assert_omp_environment;
14     # SvCUR
15     # Testing with a string
16     $output = testSvCUR("this string is 33 characters long");
17     is $output, 33, "SvCUR outputs expected value for SVPV";
18 }
19
20 done_testing(); # Automatically determines the number of tests
21
22 __DATA__
23 __C__
24
25 SV* testSvCUR(SV* input) {
26     PerlOMP_GETENV_BASIC
27     STRLEN len; // To hold the length of the scalar
28     SV* output; // To hold the output scalar
29     #pragma omp parallel private(len)
30     {
31         len = SvCUR(input); // Get the current length of the scalar
32         #pragma omp single // Ensure only one thread creates the output scalar
33         {
34             output = newSViv(len); // Create a new Perl scalar with the length as an integer
35         }
36     }
37     return output;
38 }
39 __END__

```

Figure 12: Excerpt from the scalar test harness used to exercise selected Perl SV access macros from an OpenMP parallel region while varying the requested thread count.

6.4 Safe API Functions for AV (Arrays)

Function	Description	Observed Safe?
av_len()	Returns the highest index of the array.	Yes
av_fetch()	Retrieves an element by index. In the tested pattern, concurrent reads use lval = 0 so the access does not request growth of the AV.	Conditional
av_exists()	Checks if an index exists in the array.	Yes
AvFILLp()	Fetches the last filled index in the array.	Yes
AvARRAY()	Returns a pointer to the AV's internal SV * pointer vector.	Yes

Table 6: Selected Perl array (AV) API operations and their observed behavior under the concurrent read-only tests reported in this work (Perl 5.40.0, GCC 12.2.0). “Conditional” denotes behavior that depends on avoiding structural modification of the AV.

Parameter	Description
AV *av	A pointer to the Perl array (AV) structure from which an element is to be fetched. This specifies the array to operate on.
SSize_t key	The index of the desired element in the array. Can be negative to count backward from the array's end.
I32 lval	A flag that determines whether the array should automatically grow if the specified index is out of bounds.

Table 7: Parameters to the Perl C API function `av_fetch()`, including the `lval` argument that determines whether an out-of-range access may grow the AV.

The Perl API operations listed in Table 6 produced successful results under the concurrent read-only test conditions represented in Figure 14; the full test is on GitHub²². It is important to note the case of `av_fetch`. The `lval` parameter, as shown in Table 7, plays a critical role in the behavior of `av_fetch`. When `lval` is non-zero, the function will grow the array dynamically to accommodate out-of-bounds indices, creating undefined elements as needed. This ensures that access does not fail due to array size limitations. Conversely, when `lval` is set to zero, `av_fetch` avoids modifying the array's structure, and requests for non-existent indices return `NULL`. This provides a safer way to check for existing elements without inadvertently altering the array. Under the conditions tested here, `av_fetch` behaved safely for concurrent reads when `lval` was set to 0 and no structural mutation occurred. Confidence in this access pattern led to threaded conversion functions provided by `OpenMP::Simple` via `openmp-simple.h`²³ that make it easier for a programmer to move the contents of 1D or 2D Perl arrays into corresponding native C arrays, respectively. There are versions for both `int` and `float` arrays. Figure 13 shows the two-dimensional integer case. Similarly, the tested read-only pattern can obtain the array bound with `av_len` and fetch existing elements by index inside OpenMP work-sharing code, provided no operation is allowed to resize or otherwise modify the AV.

²²<https://github.com/Perl-OpenMP/p5-OpenMP-Tests-Readonly/blob/master/t/02-array.t>

²³<https://github.com/Perl-OpenMP/p5-OpenMP-Simple/blob/master/share/openmp-simple.h>

```

1  /* 2D AoA to 2D int C array ...
2  * Convert a regular MxN Perl array of arrays (AoA) consisting of integer values, e.g.,
3  *
4  *   my $AoA = [ [qw/101 202 303/], [qw/3145 2123 892/], [qw/1917 60651 2017/] ];
5  *
6  * into a C array of the same dimensions so that it can be used as expected with an OpenMP
7  * "#pragma omp for" work sharing construct
8  */
9
10 void PerlOMP_2D_AoA_TO_2D_INT_ARRAY(SV *AoA, int numRows, int rowSize, int retArray[numRows][rowSize]) {
11     SV **AVref;
12     if (!SvROK(AoA) || SvTYPE(SvRV(AoA)) != SVt_PVAV)
13         croak("Expected Arrayref");
14     for (int i=0; i<numRows; i++) {
15         AVref = av_fetch((AV*)SvRV(AoA), i, 0);
16         if (!AVref || !*AVref || !SvROK(*AVref) || SvTYPE(SvRV(*AVref)) != SVt_PVAV)
17             croak("Expected arrayref at array[%d]", i);
18         for (int j=0; j<rowSize; j++) {
19             SV **element = av_fetch((AV*)SvRV(*AVref), j, 0);
20             if (!element || !*element || !SvOK(*element))
21                 croak("Expected value at array[%d][%d]", i, j);
22             retArray[i][j] = SvNV(*element);
23         }
24     }
25 }
26
27 /* threaded version */
28 void PerlOMP_2D_AoA_TO_2D_INT_ARRAY_r(SV *AoA, int numRows, int rowSize, int retArray[numRows][rowSize]) {
29     SV **AVref;
30     if (!SvROK(AoA) || SvTYPE(SvRV(AoA)) != SVt_PVAV)
31         croak("Expected Arrayref");
32
33     PerlOMP_GETENV_BASIC
34     #pragma omp parallel for private(AVref)
35     for (int i=0; i<numRows; i++) {
36         AVref = av_fetch((AV*)SvRV(AoA), i, 0);
37         if (!AVref || !*AVref || !SvROK(*AVref) || SvTYPE(SvRV(*AVref)) != SVt_PVAV)
38             croak("Expected arrayref at array[%d]", i);
39         for (int j=0; j<rowSize; j++) {
40             SV **element = av_fetch((AV*)SvRV(*AVref), j, 0);
41             if (!element || !*element || !SvOK(*element))
42                 croak("Expected value at array[%d][%d]", i, j);
43             retArray[i][j] = SvNV(*element);
44         }
45     }
46 }

```

Figure 13: Threaded OpenMP::Simple conversion function using `av_fetch(..., 0)` inside an OpenMP work-sharing loop to read an existing Perl array-of-arrays without requesting AV growth.

```

1  #!/usr/bin/env perl
2  use strict;
3  use warnings;
4  use Test2::V0;
5  use OpenMP;
6  use Inline (
7      C => 'DATA',
8      with => qw/OpenMP::Simple/,
9  );
10 my $omp = OpenMP->new;
11 for my $want_num_threads ( 1 .. 16 ) {
12     note "$want_num_threads threads ...";
13     $omp->env->omp_num_threads($want_num_threads);
14     $omp->env->assert_omp_environment; # (optional) validates %ENV
15     # call parallelized C function
16     # av_len
17     # Testing with an array
18     my $array_ref = [1 .. 1000 ];
19     my $output = test_av_len($array_ref);
20     is $output, scalar @$array_ref - 1, "Length of array is, as expected";
21 }
22
23 done_testing(); # Automatically determines the number of tests
24
25 __DATA__
26
27 __C__
28
29 SV* test_av_len(SV* input) {
30     PerlOMP_GETENV_BASIC // Ensures OpenMP environment variables are handled
31     AV* array;
32     int length = -1; // Default length if input is not an array
33     SV* output;
34     // Ensure the input is an array reference
35     if (SvROK(input) && SvTYPE(SvRV(input)) == SVt_PVAV) {
36         array = (AV*)SvRV(input);
37         #pragma omp parallel
38         {
39             int local_len = av_len(array); // Get the highest index in the array
40             #pragma omp critical
41             {
42                 // OpenMP ensures safe access to shared resources
43                 length = local_len;
44             }
45         }
46     }
47     // Create a new scalar to return the length
48     output = newSViv(length);
49     return output;
50 }
51
52 __END__

```

Figure 14: Excerpt from the array test harness used to exercise selected Perl AV operations from OpenMP-enabled C while varying the requested thread count.

6.5 Safe API Functions for HV (Hashes)

Not all of the functions in Table 8 behaved safely for concurrent traversal and inspection of an HV under the tested conditions. This became clear while developing the hash test harness, partially illustrated in Figure 15; the full test is on GitHub²⁴. What are called *hashes* in Perl, are known in other domains and languages as *associative arrays* or *dictionaries*. They offer constant time $O(1)$ look ups based on a hash key, and the value of each hash key is an SV (scalar value) that may point to an actual string or numerical value - or it may contain the reference of another Perl data structure of some kind (e.g., SV, AV, HV, or even a Perl code reference). Hashes are also the basis for traditional Perl Objects (via *bless*) since they are the most flexible kind of data structure Perl has to offer. Unlike arrays, hashes are unordered.

²⁴<https://github.com/Perl-OpenMP/p5-OpenMP-Tests-Readonly/blob/master/t/03-hash.t>

Function	Description	Observed Safe?
hv_fetch()	Fetches an existing value by key; the tests use the non-mutating access mode.	Yes
hv_exists()	Checks if a key exists in the hash.	Yes
hv_iterinit()	Initializes iterator state associated with the hash; unsuitable for independent concurrent traversal of the same HV in these tests.	No
hv_iternext()	Advances iterator state associated with the hash; unsuitable for independent concurrent traversal of the same HV in these tests.	No
hv_ival()	Returns an SV pointer to the value of the HE structure.	Yes
HvKEYS()	Returns the number of keys. A separate test configuration exposed a compiler/macro-expansion problem involving HeKEY; no run-time root cause is asserted here.	Yes
HvARRAY()	Returns a pointer to the hash's internal bucket array (an array of HE * bucket heads), not a copy of all hash entries. Testing revealed a situation where an inner hv_fetch caused a rehash, leading to cascading failures.	Conditional
HvMAX()	Returns the maximum bucket index in the hash's current bucket array. The number of buckets is therefore HvMAX(hv) + 1; this is not the number of elements stored in the hash.	Yes

Table 8: Selected Perl hash (HV) API operations and their observed behavior under the concurrent read-only tests reported in this work (Perl 5.40.0, GCC 12.2.0). The results describe the tested access patterns and are not a general guarantee for concurrent hash mutation or iterator use.

```

1 // Function to start parallel fetch using OpenMP in an attempt
2 // to induce some sort of memory allocation issues
3 SV* test_hv_fetch(HV *hash, AV *keys, int iterations) {
4     // Get the number of keys in the provided AV* (key array)
5     int key_count = av_len(keys) + 1; // av_len is 0-based, so we add 1 to get the total count
6     PerlOMP_GETENV_BASIC
7     // Convert the result (int found_count) to an SV (Scalar Value)
8     SV *result;
9     // Parallelize the loop using OpenMP
10    #pragma omp parallel
11    {
12        int found_count;
13        #pragma omp for
14        for (int i = 0; i < iterations; i++) {
15            found_count = 0;
16            for (int j = 0; j < key_count; j++) {
17                SV *key_sv = *av_fetch(keys, j, 0); // Fetch the key from the Perl array
18                if (key_sv) {
19                    STRLEN len;
20                    char *key = SvPV(key_sv, len); // Get the key as a C string
21                    SV **value = hv_fetch(hash, key, len, 0); // Perform hv_fetch
22                    if (value != NULL) {
23                        found_count++;
24                    }
25                }
26            }
27        }
28        #pragma omp single
29        result = newSViv(found_count); // Create a new SV from the found count (int)
30    }
31    return result;
32 }

```

Figure 15: Excerpt from the hash test harness used to exercise selected Perl HV access operations from OpenMP-enabled C while varying the requested thread count.

Two functions that provide a conventional iterative interface over a HV are `hv_iterinit` and `hv_iternext`. They maintain iterator state associated with the hash, so concurrent threads attempting to advance that shared iterator are not a suitable basis for independent traversal. The source code behind `hv_iternext` is correspondingly complex because of the bookkeeping required for hash iteration; the reader may wish to inspect this code as an exercise²⁵.

The alternative explored here uses `HvARRAY` and `HvMAX` to inspect the existing bucket structure directly. A Perl HE is a hash-entry structure: a bucket points to an HE, each HE refers to its key information and associated SV * value, and `HeNEXT` links entries that occupy the same bucket. `HvARRAY(hash)` returns a pointer to the hash's internal array of bucket-head pointers; it does not make a private copy for each OpenMP thread. `HvMAX(hash)` returns the maximum valid bucket index, so the bucket count is `HvMAX(hash) + 1`, not the number of key/value elements in the hash[9]. Consequently each thread can have a private local pointer variable while still reading the same underlying bucket array. This remains safe only while the hash structure is not mutated or rehashed. A mutating `hv_fetch` mode, autovivification²⁶, insertion, deletion, or other structural change can invalidate that assumption. Figure 16 therefore illustrates direct read-only bucket traversal rather than a general-purpose concurrent hash iterator.

During experimentation with a different arrangement of `#pragma omp parallel`, `#pragma omp for`, and `#pragma omp single`, GCC 12.2.0 produced a compilation error involving expansion of the `HeKEY` macro. The test was restructured to avoid that expansion pattern. Because the failure was not isolated beyond that configuration, this paper records the observation without attributing it to GCC, GOMP, or the Perl headers.

```

1  SV* test_hv_array(HV *hash, int iterations) {
2      int total_keys = 0;
3
4      PerlOMP_GETENV_BASIC
5
6      // Parallelize using OpenMP, each thread calls HvARRAY and iterates independently
7      #pragma omp parallel for
8      for (int i = 0; i < iterations; i++) {
9          int my_num_keys = 0;
10         HE **buckets = HvARRAY(hash); // Local pointer to the shared bucket array
11         STRLEN n_buckets = (STRLEN)HvMAX(hash) + 1;
12         for (STRLEN j = 0; j < n_buckets; j++) {
13             HE *he = buckets[j];
14             while (he) {
15                 // Read-only inspection of the existing HE would occur here.
16                 he = HeNEXT(he);
17                 ++my_num_keys;
18             }
19         }
20         int tid = omp_get_thread_num();
21         if (tid == 0) {
22             total_keys = my_num_keys;
23         }
24     }
25
26     // Return the total found count as an SV
27     return newSViv(total_keys);
28 }

```

Figure 16: Read-only traversal of an existing Perl hash bucket structure. `HvARRAY()` exposes the shared `HE **` bucket array, `HvMAX(hv) + 1` gives the number of bucket slots, and `HeNEXT()` follows entries within a bucket chain; the hash must not be structurally modified or rehashed during the traversal.

6.6 Summary of Initial Findings About Thread-Safety in the Perl API

The initial tests found a useful set of scalar access operations that behaved safely under the concurrent read-only conditions exercised here. Array and hash access required more explicit constraints because `AV` and `HV` structures can resize, rehash, autovivify, or maintain iterator state. The practical lesson is not that one Perl data type is universally thread-safe and another is not; rather, the safety of native-thread access depends on the exact API operation, the state of the value, and whether the operation can trigger interpreter-managed mutation. Explicit concurrent-read contracts in the Perl C API would make these distinctions easier for extension authors to rely upon and would broaden the set of shared-memory patterns that can be expressed safely.

²⁵<https://github.com/Perl/perl5/blob/a9edd71bc18db0f8b887fcd3249565daff946348/hv.c#L2917>

²⁶Perl's autovivification automatically creates hash keys and their associated nested structures as needed, potentially triggering a rehash of the hash's internal memory allocation to maintain efficient access.

7 The Case for Thread-Safe "Read-Only" Perl API Functions

The introduction of thread-safe *read-only* functions in the Perl API represents a critical step toward modernizing Perl's capabilities for multi-core systems while maintaining the simplicity and predictability for which Perl is renowned. This proposal focuses on a pragmatic approach that selectively integrates thread-safe functions where they provide immediate utility, without introducing unnecessary complexity to the core interpreter.

In a typical sequential programming model, operations are executed one after another to ensure data consistency and avoid race conditions. This approach works well for many use cases but increasingly limits efficiency as multi-core processors become the norm. While Perl's core has historically relied on single-threaded execution to maintain simplicity and speed, there is a clear opportunity to introduce thread-safe alternatives for *read-only* functions, in particular.

The rationale for starting with *read-only* operations is twofold:

- **Safety:** Genuinely non-mutating operations avoid write/write conflicts and are therefore the most straightforward candidates for carefully specified concurrent-read interfaces. As Section 6.1 shows, a function that appears to read may still mutate interpreter state, so the distinction must be established rather than assumed.
- **Utility:** Many computational tasks, such as traversing data structures (e.g., HVs or AVs), retrieving values, or performing look-ups, are read-intensive. Making these operations thread-safe would allow developers to parallelize common patterns without risking data corruption or unpredictable behavior.

By providing thread-safe APIs for *non-mutating* operations, Perl can enable parallel execution of common tasks in non-core components, such as extensions, libraries, or specialized computations, while leaving the single-threaded model of the Perl interpreter unchanged. This incremental approach aligns with Perl's philosophy of flexibility and backward compatibility while paving the way for future enhancements.

7.1 Benefits for Perl Programmers

Thread-safe read-only functions would offer immediate advantages to developers working with large datasets, computationally intensive tasks, or situations where performance is a priority. For example, consider the case of iterating over a large hash (HV):

- In the current model, the programmer must ensure that the hash is not accessed concurrently, leading to complex locking mechanisms or single-threaded execution.
- With thread-safe read-only functions, multiple threads could safely iterate over the same HV in parallel, significantly reducing complexity and improving performance, and since iterator order is meant to be randomized, the need to preserve some order is not present.

This enhancement would simplify multi-threaded Perl programming while ensuring that developers can rely on consistent behavior and data integrity.

7.2 Real-World Parallelism: Introducing OpenMP

While thread-safe read-only functions in the Perl API are valuable on their own, they also serve as a foundation for leveraging external parallel programming frameworks, such as **OpenMP**. OpenMP enables parallel execution by managing threads in shared-memory environments, making it particularly effective for computationally expensive tasks that can be broken into smaller, independent operations.

For instance, using OpenMP with Perl:

- A developer could parallelize read-only operations, such as traversing large datasets or computing aggregates across multiple threads.
- Complex computations that previously ran sequentially could take full advantage of multi-core processors, reducing execution time without adding manual complexity.

The preceding experiments already provide concrete examples of this pattern without requiring a hypothetical interface. Figure 13 shows an OpenMP work-sharing loop that reads existing AV elements through `av_fetch(..., 0)` without requesting array growth, while Figure 16 shows direct read-only traversal of an existing HV bucket structure under the stricter requirement that the hash not be structurally modified or rehashed. These examples illustrate the conservative design principle advocated here: keep operations that create or mutate Perl interpreter state outside worker-thread regions, and permit parallel workers to perform only those read-only accesses whose behavior has been established for the intended data state and Perl version. A future Perl API that explicitly guarantees such concurrent-read contracts would make this style substantially easier to reason about and maintain.

7.3 A Hybrid Approach Inspired by Redis

To draw a comparison, Redis²⁷ offers a well-known model for balancing sequential simplicity with selective multi-threading. Redis maintains a single-threaded core to handle most operations efficiently while selectively enabling multi-threading for specific tasks, such as I/O operations or background processing. This hybrid approach ensures that Redis remains fast and predictable for command processing, leveraging multi-threading only where it adds clear value. Perl could adopt a similar strategy:

- The main interpreter remains single-threaded to preserve simplicity.
- Thread-safe read-only functions unlock multi-threading opportunities for specific operations, particularly in non-core areas.

This method balances efficiency with maintainability, allowing developers to take advantage of multi-core systems without introducing the full complexity of parallel programming into the Perl core.

7.4 A Path Toward Multi-Core Efficiency

Introducing thread-safe *read-only* functions is not only an immediate win for developers but also a stepping stone for future improvements. By laying this foundation, Perl can:

- Enable better integration with frameworks like OpenMP for high-performance parallel computation.
- Explore additional thread-safe APIs for non-core tasks, such as network I/O or background data processing.
- Maintain backward compatibility while gradually evolving to meet modern computing demands.

In conclusion, thread-safe read-only functions provide a low-risk, high-reward opportunity to modernize Perl. They simplify the path to multi-threading, enable developers to harness the full potential of multi-core processors, and open the door to external parallel programming frameworks like OpenMP. By starting with thread-safe *read-only* operations, Perl can balance simplicity, safety, and performance, positioning itself as a more powerful and flexible tool for modern development. If the interfaces exist, the implementations can be improved later - e.g., using things like lock-free data structures. Lock-free data structures enable safe and efficient concurrent thread access without requiring locks, relying on atomic operations to avoid contention, and are a well-studied area with extensive research in computer science.

The same line of inquiry also suggests useful mutating interfaces that would require an explicit concurrent contract rather than merely a read-only one. For example, the AV tests sometimes needed to accumulate results with `av_push`; that mutation was kept out of the worker-wide access path and performed under an `omp single` region. A deliberately thread-safe append interface would address a different problem from the read-only APIs studied here, but it illustrates where future work could extend the same principle.

8 Status of the Perl+OpenMP Project, and Future Work

The Perl+OpenMP Project has developed from a set of experiments into a small, usable stack of CPAN distributions for combining Perl with OpenMP. The project has two related goals. The first is immediately practical: make established OpenMP compiler and run-time technology accessible from ordinary Perl programs without requiring each programmer to reconstruct compiler flags, run-time configuration, environment handling, and Perl/C integration machinery by hand. The second is investigative: use that working environment to identify where the Perl C API can participate safely in shared-memory native execution, and where narrowly scoped, explicitly thread-safe API interfaces would provide useful additions to Perl.

These goals reinforce one another. A practical Perl+OpenMP environment makes systematic investigation of the Perl API possible, while the results of that investigation identify safer and more idiomatic boundaries for the tools themselves. The work described in this paper should therefore not be read as claiming that the Perl C API is generally thread-safe. Rather, it seeks to identify operations whose behavior under carefully constrained concurrent use can be characterized, while advocating explicit thread-safe interfaces where such interfaces can eventually be supported by the Perl implementation.

This line of work has been public since at least 2021. An October 2021 project invitation described the goal as exploring the ways in which Perl and OpenMP can be combined idiomatically, including the specific question of

²⁷Redis is an open source, in-memory data structure store that supports a variety of data types with atomic guarantees for operations, making it suitable for caching, real-time analytics, and message brokering.

where OpenMP can be applied with respect to the Perl API and interpreter internals through `Inline::C` or `XS`.²⁸ That post also links a 2021 YAPC::Zoom lightning talk²⁹ and a two-hour Houston Perl Mongers introduction to OpenMP from Perl.³⁰ The present paper therefore represents one stage of a longer-running investigation rather than a proposal developed in response to recent discussion about Perl concurrency.

The overall effort is led by the author of this paper, Brett Estrade (OODLER on CPAN³¹). See Section 10 for acknowledgments of those who have helped shape the work.

8.1 Current CPAN Deliverables

As of September 2026, the principal Perl+OpenMP Project components are available from CPAN and form a layered tool chain.³² The versions in Table 9 describe the current project at publication time; they are *not* the versions used to generate the experimental results in this paper.

Distribution	Version	Current Project Role
<code>Alien::OpenMP</code>	0.3.10	Discovers and supplies compiler/run-time configuration for OpenMP-enabled native builds. Current portability work includes compiler-family detection and CI coverage across Linux GCC/Clang, Strawberry Perl on Windows, Apple Clang/macOS, and FreeBSD, plus GCC 10–16 sweeps.
<code>OpenMP::Environment</code>	1.5.0	Manages and validates OpenMP and GNU <code>libgomp</code> environment settings from Perl, including <code>lvalue</code> access, opt-in <code>assert/unset</code> shortcuts, and portable validation/cross-variable rules.
<code>OpenMP::Simple</code>	0.2.7	Supplies the <code>Inline::C</code> integration layer, run-time C helpers, environment-update macros, Perl/native conversion routines, and validation helpers; recent portability work also tightened where worker threads may touch Perl-managed state.
<code>OpenMP</code>	1.0.3	Provides the project metapackage and single installation point for <code>Alien::OpenMP</code> , <code>Inline::C</code> , <code>OpenMP::Environment</code> , <code>OpenMP::Simple</code> , and the <code>perlomp</code> documentation.
<code>perlomp</code>	0.01	Provides the Perl-style <code>perldoc perlomp</code> manual describing installation, project components, run-time initialization, portability, environment validation, native-thread safety considerations, and introductory usage patterns.

Table 9: Current Perl+OpenMP Project deliverables on CPAN as of September 2026. These versions document the present state of the project and must not be confused with the Perl 5.40.0/GCC 12.2.0 experimental environment used for the results reported in this paper.

The stack now gives the project a clearer separation of responsibilities. `Alien::OpenMP` addresses compiler and run-time discovery; `OpenMP::Environment` provides the Perl-side run-time configuration model; `OpenMP::Simple` provides native integration helpers; `OpenMP` installs the pieces together; and `perlomp` provides a common manual. The practical result is that the mechanics required to begin experimenting with Perl and OpenMP no longer have to be reconstructed for each program.

8.2 How the Current Project Fulfills the Thesis of This Paper

The current distributions implement the practical side of the thesis: OpenMP can be made available to Perl programmers while maintaining a deliberate boundary between the Perl interpreter and native OpenMP worker threads. The evolution of `OpenMP::Simple` provides a concrete example. In version 0.2.7, its threaded string-conversion path was revised so that Perl API access and allocation occur on the calling Perl thread, while OpenMP workers perform only native-buffer copying. This is not merely an implementation detail; it illustrates the conservative strategy advocated throughout this paper: determine exactly which work must remain under Perl’s ownership and parallelize only the work that can safely occur beyond that boundary.

The experiments in this paper investigate the complementary case. Some read-only Perl API operations behaved safely, under the conditions tested here, when used directly by multiple OpenMP worker threads without first converting the entire Perl data structure into a native representation. Where that behavior can be characterized reliably, it can reduce conversion boilerplate and permit more idiomatic interfaces between Perl and parallel C code. These observations do not constitute a general thread-safety guarantee for the Perl API. They instead motivate the

²⁸https://blogs.perl.org/users/oodler_577/2021/10/open-invitation-to-participate-in-perl-openmp-on-github.html

²⁹https://www.youtube.com/watch?v=LWs_rJT9lg4

³⁰<https://www.youtube.com/watch?v=wHjmxGJd7rQ>

³¹<https://metacpan.org/author/OODLER>

³²<https://www.cpan.org/authors/id/O/OODLER/>

longer-term project goal of identifying useful operations for which explicit concurrent semantics could reasonably be documented or provided by Perl.

This creates a useful feedback loop between software delivery and research. Current `Perl+OpenMP` Project releases can become more conservative where the boundary is known to be sensitive, as the string-conversion change demonstrates. At the same time, the test environment supplied by the project makes it easier to investigate candidate Perl API operations and to distinguish genuine read-only access from apparently read-only calls that can mutate interpreter state. The project therefore occupies a position between application tooling and Core research: the CPAN distributions make present techniques usable now, while the continuing API investigation asks what Perl itself might eventually guarantee.

The broader compiler and platform coverage now present in `Alien::OpenMP` and `OpenMP::Simple` is an important improvement in the project’s practical reach, but it is not evidence that the Perl C API experiments in this paper have been repeated under newer compilers, newer Perl releases, or non-GNU OpenMP run-times. Those replications remain separate future work.

8.3 Contemporaneous Perl Core Discussion

While this manuscript was being revised for publication in 2026, a September 2026 discussion on `perl5-porters` proposed several core interfaces for concurrency and multi-core execution, including a “multicore XS” release/acquire bracket and an asynchronous offload interface.³³ The proposed bracket deliberately places a conservative boundary around native work: the bracketed section is intended to execute as pure C without accessing Perl interpreter data. That design addresses a broader and complementary problem to the one studied here.

The experiments in this paper ask a narrower question: whether particular, already-established Perl data structures admit useful concurrent read access through specific Perl C API operations when no interpreter-managed mutation is triggered. The two approaches are therefore not contradictory. A “do not touch Perl data” rule is an appropriate general-purpose boundary for unconstrained native work, while an explicitly documented concurrent-read contract could permit carefully selected operations beyond that boundary. The experiments and conclusions reported here predate the September 2026 discussion; this subsection is included only to place the work in the context of a newly active core conversation and does not alter the historical experimental environment or the claims made from it.

A related `perl5-porters` discussion in December 2024 considered documentation and semantics surrounding `MULTIPLICITY`, embedded interpreters, and concurrent execution.³⁴ That topic concerns multiple interpreter contexts and is distinct from the same-data, same-process concurrent-read patterns investigated here, but it illustrates a broader continuing interest in making Perl’s concurrency boundaries more explicit.

8.4 Continued Exploration of Read-Only Perl API Functions and Macros

This paper has only begun to peel back the layers present in the Perl API. Continued work should catalogue read-oriented functions and macros, identify which ones remain non-mutating under precisely stated conditions, and reproduce candidate access patterns across relevant Perl and toolchain versions before stronger contracts are proposed. This will lead to a better understanding of how to integrate Perl and OpenMP in efficient and idiomatic ways without turning observations from one configuration into universal claims. A preliminary set of functions for reading scalar, array, and hash values has been presented here; there are many more functions and macros in the Perl C API to characterize.

8.5 Performance Characterization

The experiments reported in this paper are correctness and stress investigations of the Perl C API in OpenMP parallel regions; they are not intended to constitute a comparative benchmark suite. Establishing the interfaces and safe usage patterns comes first, because performance measurements are most useful once the data representation and synchronization rules are well understood. The work presented here nevertheless creates a useful basis for later empirical characterization across workload size, native versus Perl data representation, thread count, compiler and OpenMP run-time, and hybrid process/thread execution models. Such measurements can be undertaken without changing the central purpose of the present paper or presuming in advance which approach will be fastest for a particular workload.

³³<https://www.nttp.perl.org/group/perl.perl5.porters/2026/09/msg271161.html>

³⁴<https://www.nttp.perl.org/group/perl.perl5.porters/2024/12/msg269321.html>

8.6 Cookbook

A natural next step from this work is to develop a set of idioms and best practices and publish them on CPAN as the **Perl+OpenMP Project Cookbook**. The intended audiences approach the combination of Perl and OpenMP from different directions:

1. Those familiar with OpenMP and C, but not with Perl
2. Those familiar with Perl and possibly C, but not OpenMP
3. Those familiar with both Perl and OpenMP
4. ~~Those familiar with neither Perl nor OpenMP~~

The last group is not a primary target; readers need some grounding in Perl or OpenMP before a cookbook can be effective. Documentation should nevertheless be sufficient for readers familiar with Perl, OpenMP, or both to get started quickly. The **Inline::C Cookbook**³⁵ is a useful model, and the **Perl+OpenMP Project Cookbook** should feel like a companion to it. Perl programs can also target HPC batch systems such as Slurm, including scripts that prepare and submit threaded jobs. Improving the user experience for shared-memory Perl programs on HPC systems should be a fruitful area for future work.

8.7 Expand **OpenMP::Simple** to Support Idioms and Best Practices

The whole point of **OpenMP::Simple** is to make it as streamlined as possible for anyone in the aforementioned user classes to focus on what they want to do rather than on integration boilerplate. Perl developers like how quickly Perl gets out of their way when working on a task, so that is the goal with **OpenMP::Simple**. It still remains to be seen which Perl utility methods, C functions, and C macros are most useful to provide. A large part of determining this will come from continuing the work presented in this paper and getting other people involved to try out the ideas they may have.

To enhance the functionality of **OpenMP::Simple**, several conversion functions and verification macros have been developed to support idiomatic Perl-to-C interoperability. These functions facilitate seamless data transformation while ensuring type consistency across both single-threaded and multi-threaded environments. The tables below summarize the available conversion functions and verification macros.

Function Name	Description
<code>PerlOMP_1D_Array_TO_1D_FLOAT_ARRAY</code>	Converts a 1D Perl array to a C array of floats.
<code>PerlOMP_1D_Array_TO_1D_FLOAT_ARRAY_r</code>	Threaded version: Converts a 1D Perl array to a C array of floats.
<code>PerlOMP_1D_Array_TO_1D_INT_ARRAY</code>	Converts a 1D Perl array to a C array of integers.
<code>PerlOMP_1D_Array_TO_1D_INT_ARRAY_r</code>	Threaded version: Converts a 1D Perl array to a C array of integers.
<code>PerlOMP_1D_Array_TO_1D_STRING_ARRAY</code>	Converts a 1D Perl array to a C array of strings.
<code>PerlOMP_1D_Array_TO_1D_STRING_ARRAY_r</code>	Threaded version: Converts a 1D Perl array to a C array of strings.
<code>PerlOMP_2D_AoA_TO_2D_FLOAT_ARRAY</code>	Converts a 2D Perl array of arrays to a 2D C array of floats.
<code>PerlOMP_2D_AoA_TO_2D_FLOAT_ARRAY_r</code>	Threaded version: Converts a 2D Perl array of arrays to a 2D C array of floats.
<code>PerlOMP_2D_AoA_TO_2D_INT_ARRAY</code>	Converts a 2D Perl array of arrays to a 2D C array of integers.
<code>PerlOMP_2D_AoA_TO_2D_INT_ARRAY_r</code>	Threaded version: Converts a 2D Perl array of arrays to a 2D C array of integers.
<code>PerlOMP_2D_AoA_TO_2D_STRING_ARRAY</code>	Converts a 2D Perl array of arrays to a 2D C array of strings.
<code>PerlOMP_2D_AoA_TO_2D_STRING_ARRAY_r</code>	Threaded version: Converts a 2D Perl array of arrays to a 2D C array of strings.

Table 10: Perl-to-C array conversion functions available in **OpenMP::Simple** as used in this work, including serial and “_r” threaded variants for one- and two-dimensional data.

The functions in Table 10 extend **OpenMP::Simple**’s capabilities by providing explicit data conversion across different data types and dimensions. The functions handle both 1D and 2D Perl arrays, converting their values into native C arrays that are directly suited to conventional OpenMP numerical kernels. Whether that conversion improves end-to-end performance depends on the workload and on how often the Perl/C boundary is crossed; the

Macro Name	Description
<code>PerlOMP_VERIFY_1D_Array</code>	Checks if a Perl variable is a valid 1D array reference.
<code>PerlOMP_VERIFY_2D_AoA</code>	Checks if a Perl variable is a valid 2D array of arrays reference.
<code>PerlOMP_VERIFY_1D_FLOAT_ARRAY</code>	Verifies that the 1D array contains only floats.
<code>PerlOMP_VERIFY_1D_INT_ARRAY</code>	Verifies that the 1D array contains only integers.
<code>PerlOMP_VERIFY_1D_DOUBLE_ARRAY</code>	Verifies that the 1D array contains only doubles.
<code>PerlOMP_VERIFY_1D_STRING_ARRAY</code>	Verifies that the 1D array contains only characters (strings).
<code>PerlOMP_VERIFY_2D_FLOAT_ARRAY</code>	Verifies that the 2D array of arrays contains only floats.
<code>PerlOMP_VERIFY_2D_INT_ARRAY</code>	Verifies that the 2D array of arrays contains only integers.
<code>PerlOMP_VERIFY_2D_DOUBLE_ARRAY</code>	Verifies that the 2D array of arrays contains only doubles.
<code>PerlOMP_VERIFY_2D_STRING_ARRAY</code>	Verifies that the 2D array of arrays contains only strings.

Table 11: Array-reference and element-type verification macros available in `OpenMP::Simple` as used in this work, intended to validate Perl inputs before native-array conversion.

intended idiom is to convert once around a sufficiently substantial native computation rather than repeatedly inside its inner loop. The “_r” suffix denotes a threaded version of the conversion function.

Additionally, verification macros in Table 11 provide type checks before conversion. These macros reduce errors due to mixed or unexpected data types and make the Perl-to-native boundary explicit. The expansion of `OpenMP::Simple` with these idioms and best practices is intended to improve compatibility and reliability when integrating Perl with OpenMP, while leaving quantitative performance characterization to workload-specific measurements.

8.8 Open Areas - Perl As A Memory Allocator, OpenMP Direct Memory Access

A related area for future exploration is the use of Perl as the owner or allocator of memory that native code then treats as an explicitly defined buffer. For example, `Task::MemManager`, created by Christos Argyropoulos (CHRISARG³⁶), provides interfaces to alternative memory allocators for C- or assembly-based work invoked from Perl. A Perl value can serve as the handle through which native code reaches a reserved region, provided ownership, lifetime, representation, and synchronization rules remain explicit. For OpenMP this suggests another useful boundary: establish or allocate the native buffer while under Perl’s control, perform parallel work on the buffer without manipulating unrelated Perl interpreter metadata, and expose the results back through a well-defined Perl-facing interface. This direction extends the boundary-oriented approach discussed in Section 5 and warrants separate testing.

9 Conclusion

The findings of this paper highlight an under-explored opportunity within the Perl C API: the possibility of safely performing selected read-only accesses to Perl data structures from multi-threaded native code under carefully constrained conditions. The experimental evidence is deliberately narrow: it comes from Perl 5.40.0 built with GCC 12.2.0 and GNU `libgomp`, and later improvements to the `Perl+OpenMP` Project toolchain do not retroactively widen those results. What the later project work does provide is a more usable and increasingly portable framework in which the boundary between interpreter-owned operations and native parallel work can continue to be studied.

The `Perl+OpenMP` Project now supplies practical CPAN tooling for this work through `Alien::OpenMP`, `OpenMP::Environment`, `OpenMP::Simple`, the `OpenMP` metapackage, and the `perlomp` manual. Those distributions do not themselves make the Perl C API thread-safe. Instead, they reduce the integration burden, encode increasingly conservative boundary patterns where appropriate, and make it easier to reproduce and extend the investigation. In parallel, the research argument of this paper remains that Perl would benefit from carefully specified concurrent-read interfaces for operations whose semantics can genuinely support them, with explicit thread-safe variants considered for selected mutating operations where there is sufficient value and a sound implementation strategy.

That combination of usable tooling now and evidence-driven Core advocacy later is the central contribution of the project. It allows Perl programmers to exploit mature shared-memory technology without pretending that arbitrary interpreter state is safe to access concurrently, while creating a concrete path for Perl’s native API contracts to become more explicit where modern multi-core use cases justify the effort.

³⁵<https://metacpan.org/dist/Inline-C/view/lib/Inline/C/Cookbook.pod>

³⁶<https://metacpan.org/author/CHRISARG>

10 Acknowledgments

The author would like to thank Nerdvana, x86, Tony Cook, Mohawk, Dr. Christos Argyropoulos, and bbrtj for their invaluable guidance and ongoing support throughout this work. Their insights and expertise have been instrumental in shaping the direction of this research, and their contributions are deeply appreciated. The author is especially grateful to Dr. Christos Argyropoulos for his thoughtful review of this manuscript and beneficial comments. His review sharpened the treatment of copy-on-write behavior under `fork`, highlighted the practical interaction between Perl process parallelism and OpenMP run-time threading, prompted a more precise description of Perl `AV` and `HV` internal memory structures, and called attention to FFI-loaded OpenMP libraries as another important path by which native threading can enter a Perl process.

References

- [1] Frederick P. Brooks Jr. *The Mythical Man-Month: Essays on Software Engineering*. Addison-Wesley, Reading, Massachusetts, 1st edition, 1975.
- [2] GNU Project. Gomp - gnu offloading and multi processing, 2024. Accessed: 2024-06-04. URL: <https://gcc.gnu.org/projects/gomp/>.
- [3] GNU Project. Gcc 12 release series: Changes, new features, and fixes, 2022. OpenMP 5.0 support was extended and selected OpenMP 5.1 features were added in GCC 12. URL: <https://gcc.gnu.org/gcc-12/changes.html>.
- [4] Michael Kerrisk and the Linux man-pages project. `fork(2)` — create a child process. Linux implements `fork()` using copy-on-write pages. URL: <https://man7.org/linux/man-pages/man2/fork.2.html>.
- [5] Tim Jenness and Simon Cozens. *Extending and Embedding Perl*. Manning Publications, 2002. A comprehensive guide to expanding Perl’s functionality and integrating it with C programs.
- [6] Christos Argyropoulos. Enhancing non-perl bioinformatic applications with perl: Building novel, component based applications using object orientation, pdl, alien, ffi, inline and openmp, 2024.
- [7] OpenMP Architecture Review Board. Openmp application programming interface, version 5.0, 2018. See the resource-pause run-time routines `omp_pause_resource` and `omp_pause_resource_all`. URL: <https://www.openmp.org/spec-html/5.0/openmpsui153.html>.
- [8] Chunhua Liao, Oscar Hernandez, Barbara Chapman, Wenguang Chen, and Weimin Zheng. Openuh: An optimizing, portable openmp compiler. *Concurrency and Computation: Practice and Experience*, 19(18):2405–2423, 2007.
- [9] Perl 5 Porters. perlapi — autogenerated documentation for the perl public api, perl 5.40.0, 2024. Documentation for `AvARRAY`, hash handling, and Perl C API data structures. URL: <https://perldoc.perl.org/5.40.0/perlapi>.