Document Number: N3556
Date: 2013-03-18

Pablo Halpern, Intel Corp.
pablo.g.halpern@intel.com

Charles E. Leiserson, MIT
cel@mit.edu

Thread-Local Storage in X-Parallel Computations

Abstract

The C and C++ Standards Committees are considering several proposals for adding parallelism to their respective languages and/or libraries, especially to support fork-join parallelism [N3409] and vector units [N3419]. It is prudent, naturally, to consider the impact of any linguistic proposal on the semantics of existing features. In this document, we consider specifically the impact of parallelism on the semantics of thread-local storage (TLS). Although our discussion is generic and not dependent on any particular parallel model, it is convenient to have a stand-in name for a particular model. To this end, we designate the model under discussion as being an ***X-parallel*** model, where X might represent a variety of individual parallelization technologies, including vector units, GPU's, task-based multithreading, attached processing, and the like.[1] Fundamental to the discussion is the notion that, just as threading augments a single process with concurrency, an X-parallel model augments a single thread with parallelism. That is, we are interested in models that *augment* threading, not models that *compete with* threading.

The purpose of this paper is to develop terminology so that the impact of any X-parallel model on TLS can be described clearly and evaluated effectively. We compare the semantics of accessing a TLS variable within an X-parallel region to the semantics of accessing the same variable in serial execution. We propose a 5-level hierarchical

[1]We are not suggesting that the Standards Committees consider standardizing linguistics and libraries for all these technologies. Rather, it is our belief that a generic discussion helps to separate those issues that transcend any particular X-parallel model from those issues that are model specific.

taxonomy of ***TLS concordance***, where each level in the hierarchy is more faithful to the way the existing threading model treats TLS than the level below. Placing an X-parallel model lower within the taxonomy does not necessarily mean that it is a bad model, but lower in the hierarchy does mean that an X-parallel model is a less faithful extension to the existing standard than a higher-level model, at least as far as its adherence to legacy TLS behavior is concerned.

We conclude with a brief discussion in which we advocate that any X-parallel model should provide Level-5 (full) concordance — the highest level — unless there is a compelling reason to the contrary.

Contents

1. An evolutionary view of thread-local storage

Long before thread-local variables were invented, programming languages had ***global*** variables: variables with unlimited scope. But global variables are not truly global because they are local to the process in which they are defined; each separate process has its own global variables. Using today's terminology, we might rightly refer to them as "process-local variables." Moreover, within a process execution was serial.

Eventually, multithreading facilities were added to allow single processes to be concurrent. Instead of consisting of a serial chain of executing instructions, a process now consists of a collection of serial ***threads*** sharing the same address space. A new-style process containing only one thread continues to act like an old-style process, but the shared-memory multithreading model tore down the protective walls between cooperating threads in a single process. Each thread has its own execution stack, but any thread in a process can access any memory address in the process, including each others' stacks and global variables.

The ability of multiple concurrent threads to share data, especially global data, meant that the problem of ***race conditions*** had to be addressed. Multiple concurrent threads accessing a global variable create a ***race*** if at least one of those threads writes to the variable. Two solutions emerged: synchronization (mutex locks, condition variables, and the like) and TLS. In both cases, the burden of using these facilities to prevent races and ancillary anomalies such as deadlocks falls to the programmer.

TLS exhibits particular advantages over synchronization, not the least of which is that it avoids communication overheads. Since each thread maintains its own version of a thread-local variable, races on an otherwise global variable can be avoided. (It is possible for two threads to race on a thread-local variable if the address of the thread-local variable is passed from one thread to another, but this is rare.) A thread can typically access its own local version of a variable, which is guaranteed to be different from the local version of the variable in another thread. By providing local copies of variables, TLS avoids unintended communication without requiring radical restructuring of code when it is threaded.

One unfortunate fact about TLS is its name. Thread-local variables actually have more in common with global variables than with local variables. Unlike stack-based local variables, which are well structured and whose scope and lifetime are tied to a block, thread-local variables are unstructured. They have global scope and visibility, as well as comparatively messy lifetimes. It probably would have been better to use a term like "thread-specific global storage" rather than "thread-local storage," but this nomenclature ship has sailed, and changing terms would likely only compound confusion.

With the advent of vector units, multicore computing, graphical processing units, and attached parallel processors, the problem of incorporating X-parallel computations within a single thread has emerged. It is plain that some parallel models, such as vector units, are X-parallel models, because they clearly parallelize a single instruction stream using a SIMD model, but are clearly different from threads. It may be confusing to some, however, that other parallel models, such as some forms of task parallelism, are also X-parallel models. The confusion stems from the fact that existing implementations of tasking are usually built using multiple OS threads as "workers." How can a model that is built using multiple threads — multiple instruction streams — be viewed as parallelizing a single thread?

The answer lies in abstraction. Cilk [MIT], for example, ensures that every parallel computation has ***serial semantics***: it can always be executed as a single linear instruction stream, even though it can also be executed in parallel. Moreover, Cilk — as well as other task-parallel models such as Habanero, OpenMP, TBB, TPL, and X10, to name a few — can be implemented without OS threads. Indeed, there exists at least one

implementation of Cilk technology that does not bind Cilk "workers" to OS threads. Threads provide a particular implementation technology, which is a technical detail and not fundamental to this particular task model.

Of particular relevance, the proposal for fork-join parallelism [N3409] we are developing for the C++ Standards Committee regards an X-parallel model. That is, our proposal aims to *augment* threads, not *compete with* them. Similarly, the proposal for vector-unit parallelism [N3419] and various proposals for parallel libraries [N3408 and N3429] are also X-parallel models. Thus, whatever the X-parallel model, it behooves us to understand how X-parallel models interact with the legacy C++ TLS model generically.

2. Nomenclature for describing X-parallel computations

It is helpful to agree on some basic nomenclature for X-parallel computations. An X-parallel computation can be described in terms of an ***execution DAG*** (directed acyclic graph) that describes the ordering constraints among operations. In such a DAG (see Figure 1), two operations *a* and *b* relate in exactly one of the following ways:

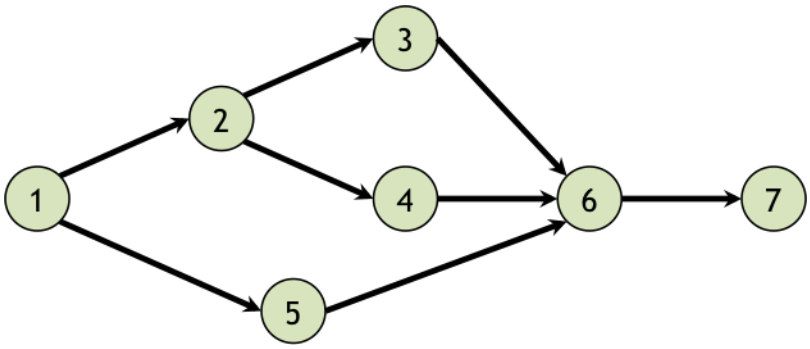

Figure 1: An execution DAG for an X-parallel computation. Circles represent operations, and arrows between them represent scheduling dependencies.

1. a = b: They are the same operation.
2. a ≺ b (a precedes b): There exists a directed path through the DAG from a to b (but not vice-versa). Operation a must execute before operation b.
3. a ≻ b (a follows b): Equivalent to b ≺ a. Operation a must execute after operation b.
4. a ∥ b (a parallels b): No directed path exists in the DAG from a to b nor does one from b to a. The logical order of a and b is indeterminate.

The DAG from Figure 1 shows the following relationships, among others:

$1 \prec 2$
$2 \parallel 5$
$3 \parallel 5$
$6 \succ 5$
$1 \prec 6 \prec 7$

The execution DAG is dynamic and can be different for each set of inputs to a program. For example, a vector or parallel loop with iterations $I_0, I_1, I_2, \ldots, I_{n-1}$ might have an execution DAG that looks something like **Figure 2**, where the structure depends on the value of n for the specific execution.

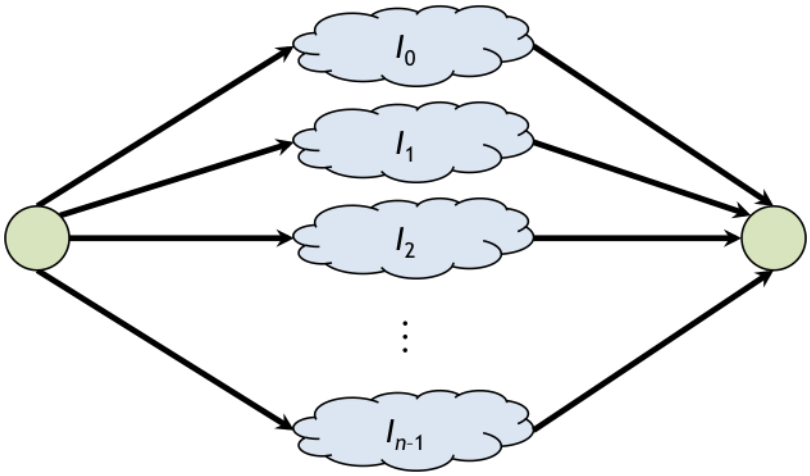

***Figure 2**: An execution DAG for a vector or parallel loop*

3. Characterizing TLS accesses within X-parallel regions of code

When a new X-parallel model is proposed, we contend that the proposers should specify how the new paradigm interacts with existing global and thread-local storage. In this section, we propose a taxonomy of "concordance" as a multi-level hierarchy that describes how an X-parallel model interacts with the existing standard for thread-local storage. Our taxonomy is not exhaustive, and any X-parallel model must address additional concerns depending on the level of concordance it purports to support.

If a user creates multiple separate threads, accesses to a *single* thread-local variable V yield *different* objects in each thread. The ***value*** of V is not of interest here, but its ***identity*** in each thread (usually, different identities are detectible by V having a different address in each thread). For the purpose of this discussion, we'll use the notation V_x to describe the identity of V at a specified location x in the program.

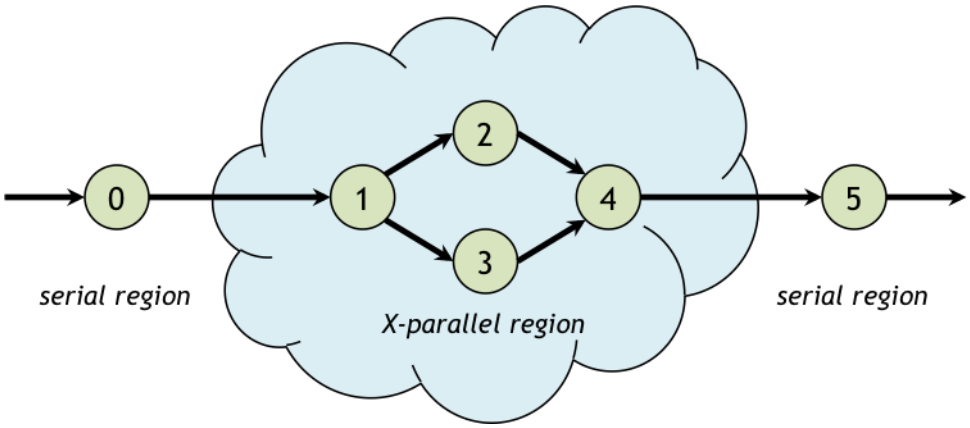

Figure 3: A thread containing an X-parallel region.

Recall that our goal is to understand how TLS should operate when a thread is augmented with an X-parallel region. Figure 3 illustrates a portion of a thread containing an X-parallel region for some unspecified parallelism technology X. A thread local variable *V* may be accessed at up to six points within the thread, labeled 0, 1, …, 5.

Suppose that the computation accesses V before and after the X-parallel region — namely, at points 0 and 5 — and that TLS accesses to V within the X-parallel region are forbidden. If the access to *V* at point 5 always yields the identical object as at point 0 ($V_5 \equiv V_0$), then the X-parallel model has achieved a minimal level of ***TLS concordance***. In other words, the meaning of TLS accesses at points 0 and 5 when using the X-parallel technology is consistent with a serial implementation of the intervening computation in which the X-parallel technology is not used. As we shall see, we describe this minimal degree of thread concordance as "Level 1" concordance. We believe that every X-parallel model should support at least Level 1.

Suppose now that the computation accesses V at points 0 and 1 with no other accesses to V within the X-parallel region. Considering point 1, the X-parallel model may support one of three possibilities:

- The TLS access to *V* at point 1 is ***forbidden*** (that is, ill-formed or undefined).
- The TLS access to *V* at point 1 is ***discordant*** in that the object V_1 produced is not guaranteed to be identical to (have the same identity as) V_0.
- The TLS access to *V* at point 1 is ***concordant*** in that the object V_1 produced is identical to V_0, that is, $V_2 \equiv V_0$.

It is fair to say that if one TLS access within the X-parallel region is forbidden, so are multiple accesses. If multiple accesses are permitted, however, the behavior may depend on whether the accesses are in series or in parallel.

For example, suppose that the multiple TLS accesses that occur within the X-parallel region are all in series. For example, in Figure 3, the accesses might occur at points 1 and 2, points 1 and 4, or points 1, 2, and 4. Since these accesses are all in series, we treat them as concordant only if all of the individual TLS accesses are concordant, and otherwise discordant.

The behavior for parallel TLS accesses may be different than for TLS accesses that are in series. For example, a vector model may allow one lane to access TLS but forbid multiple lanes from doing so. The behaviors can be grouped into a hierarchy of concordance levels, where each higher-numbered level provides stronger guarantees than lower-numbered levels, as shown in Table 1.

		Parallel access in X-parallel region		
		forbidden	**discordant**	**concordant**
Serial-only access in X-parallel region	**forbidden**	Level 1		
	discordant	Level 2	Level 3a	
	concordant	Level 3b	Level 4	Level 5

Table 1: Hierarchy of concordance levels in an X-parallel region.

Thus, Level-1 concordance provides the minimum thread concordance described earlier, guaranteeing only that $V_0 \equiv V_1$, and level 5 provides the maximum thread concordance, guaranteeing that $V_0 \equiv V_1 \equiv V_2 \equiv V_3 \equiv V_4$. Levels 3a and 3b are incomparable; neither has strictly stronger guarantees than the other.

Each level shown in Table 1 is characterized in more detail in Table 2.

Level	serial access	parallel access	Description	Example
1	forbidden	forbidden	No TLS access is allowed within the X-parallel region.	An attached or GPU model that does not have access to TLS.
2	discordant	forbidden	Serial access to TLS within the X-parallel region is allowed, but the identity of the object may differ from the serial region.	An X-parallel model that always runs in a special thread.

Level	serial access	parallel access	Description	Example
3a	discordant	discordant	Full access to TLS is allowed, but the identity of the object within the parallel region may differ from the serial region.	A model that exposes the use of threads for parallel computation (`std::async` is such a model).
3b	concordant	forbidden	Serial access to TLS is permitted, but parallel accesses are not.	A vector model where only the first lane may access TLS.
4	concordant	discordant	Serial access to TLS within the parallel region is concordant with the serial region, but parallel accesses to TLS may identify different objects.	A vector model in which the first lane gets the concordant view, and the others have their own view — a "merging" of X-local and thread-local storage.
5	concordant	concordant	All TLS accesses in the parallel region produce the same identity as in the serial region.	Most vector models (all lanes run in the same thread).

Table 2: Characteristics of each TLS concordance levels

The easiest levels to reason about appear to be Level 1 (no TLS accesses are permitted in an X-parallel region) and Level 5, what we call ***full concordance***. An advantage of full concordance over Level-1 concordance is that it makes the X-parallel model ***modeless*** with respect to TLS — the programmer need not distinguish whether a TLS access belongs to an X-parallel region, which has implications for modularity and composability. Table 2 gives examples where other levels might show up in practice. Because global (i.e., process-local) variables have the same identity in every thread within the process, the C++ (as well as POSIX and Windows) threading model can be said to implement full concordance for process-local storage.

Although it might seem that Level-5 concordance is the ideal to strive for, it is not always practical. Full TLS concordance is natural for vector parallelism, but it would be hard to implement for X-models that involve offloading computations to separate coprocessors or GPGPU's. For strict fork-join models like `parallel_for`, `parallel_invoke`, and `cilk_spawn`, full TLS concordance can be attained, but users should be discouraged from thinking of workers as threads, since they may not be implemented as threads in all cases.

4. Races and X-local storage

In addition to specifying the level of TLS concordance, an X-parallel model should also specify the expected results of reads and writes to a thread-local variable, especially with respect to potential races among TLS accesses.

- If an access is read-only, then no races can occur, regardless of thread concordance level.
- For Levels 1, 2, and 3b, no races can occur because parallel accesses to a TLS variable are forbidden.
- For Levels 3a and 4, the X-parallel model should specify whether read-write or write-write accesses to the same thread-local variable in parallel might create a race. If so, the X-parallel model might provide X-locks or X-local storage (see below) to mitigate such a race.
- For Level 5 (full TLS concordance), parallel read-write or write-write accesses to a TLS variable can create a race. Again, the X-parallel model may provide facilities to mitigate such races.

To help avoid races, X-parallel models may provide **X-local storage** specific to the model. Generally, X-local storage is tied to the conceptual ***owner*** of the storage — usually the entity to which the variable's lifetime is tied. Table 3 lists some examples of X-local storage.

Name	Owner	Model
simd-lane local	a lane within a CPU's vector unit	Vector parallelism (single-instruction stream, multiple-data stream)
gpu-thread local	a single GPU thread	GPGPU parallelism
warp local	jointly owned by all threads in a warp	GPGPU parallelism
worker local	a worker (scheduling) thread	Fork-join parallelism
task local	a task	Fork-join parallelism or multithreaded concurrency
attached local	an attached computing device	Attached processing unit

Table 3: Examples of X-local storage

5. Implementation concerns

In order to achieve a high level of TLS concordance, it may be necessary to make `std::thread` a slightly thicker layer around the underlying OS thread than would otherwise be needed, in order to accommodate parallelism technologies that are implemented on top of OS threads. For example, within the Intel® Cilk™ Plus scheduler, the workers that are managed by the scheduler are usually implemented using OS threads, but to achieve full TLS concordance, they must be treated as X-parallel components of the initial (serial) thread. Since a thread ID is just a read-only thread-local variable, if TLS is correctly handled by the scheduler, then all X-parallel subparts can return the same thread ID.

Intel® Cilk™ Plus has given us considerable experience with fork-join parallelism and vector parallelism, both areas that the committee hopes to standardize in the C++17 time frame. With no implementation effort, vector code naturally achieves full concordance. That is, any access to a TLS variable within a vector-parallel region of code accesses the identical object as an access in the serial portion of the same thread. We have found it desirable to also support full concordance within the fork-join technology. With remarkably little effort, we were able to implement a library class that modeled the desired behavior of TLS, complete with lazy construction and end-of-thread destruction. Since we did not have a standard-compliant implementation of `thread_local` to work with at the time, nor were we working with compiler sources, we were not able to investigate whether a compiler implementing this model could retain binary link-time compatibility with object files compiled before such a change.

We recognize that binary compatibility and ABI stability are important and that there are challenges involved in this transition. Similarly, there were challenges involved when threads were first introduced as a layer on top of the operating system. There was a time, for example, when making a blocking I/O call from any thread would block the entire process. Eventually, however, thread facilities were moved into the OS proper, and the runtime libraries got a little thinner. Although it is critical that everything we propose be implementable and efficient, we believe that any short-term difficulties will be ameliorated as time produces more parallel-aware operating systems.

6. Recommendations

When discussing any parallel extension to C++, regardless of the X-parallel model, its interaction with TLS must be considered and specified. Failure to do so can result in an incomplete standard or, worse, result in unnecessary anxiety within the standards Committee such that no parallelism proposal is accepted into the standard.

In this paper, we have presented a vocabulary and taxonomy for cleanly describing the interactions between TLS and X-parallel computations. We recommend that this terminology be used to inform discussions of the various X-parallelism proposals that are before the Committee.

Specifically, we advocate that a proposal to add an X-parallel facility to C++ answer the following questions:

- Does the X-parallel model meet the minimum concordance guarantee that a TLS access after an X-parallel computation refers to the same object as an access before the X-parallel computation?
- What level of thread concordance does the X-parallel model offer for TLS?
- What restrictions does the X-parallel model impose on TLS accesses? For example, the model might forbid writing to TLS in parallel.
- If races are possible on TLS variables, how can they be resolved or avoided?
- If logical and practical, are there new types of X-local storage that should be introduced to support new X-parallelism models?

An X-parallel model can be useful and easy to reason about even if it supports a low level of thread concordance with respect to TLS. An X-parallelism proposal should be precise about its assumption about TLS concordance, and it should otherwise provide enough details to assure the Committee that the model is consistent with the rest of the standard.

7. Acknowledgments

Thanks to all who reviewed earlier drafts and provided feedback.

8. References

N3487 *TLS and Parallelism* (presentation to SG1, 2012-05-08)

N3409 *Strict Fork-Join Parallelism*, Pablo Halpern, 2012-09-24

N3419 *Vector loops and Parallel Loops*, Robert Geva, 2012-09-21

MIT *The Cilk Project Home Page*

N3408 *Parallelizing The Standard Algorithms Library*, J. Hoberock, O. Giroux, V. Grover, J. Marathe, et al., 2012-09-21

N3429 *A C++ Library Solution To Parallelism*, A. Laksberg, H. Sutter, A. Robison, S. Mithani, 2012-09-21