# TLS and Parallelism

N3487=12-0177

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

Presented to SG1 (Parallelism)

2012-05-08

# Abstract

I present three different realistic use cases for thread-local storage (TLS). Using examples in PPL, Cilk, and TBB, I show that no one definition of the `thread_local` storage class would suffice for all three use cases, regardless of parallelism platform. We thus need to broaden our support for TLS, with new language and/or library features, in order to allow parallel programming to co-exist with the different uses of TLS. Finally, I propose a specific meaning for the existing keyword, `thread_local`, matching one of the three use cases.

# Parallelism Terminology

- **Task**: A unit of work that can be scheduled and executed asynchronously. Examples of tasks:
  - Each iteration of a parallel loop
  - The invokable argument of `std::async`
  - The branches of a `parallel_invoke` in PPL or TBB
  - The continuation of a `cilk_spawn` – I.e., the code that runs between the `cilk_spawn` and the corresponding `cilk_sync`
- **Worker**: The member of a thread pool that executes a task. Worker threads are typically managed by a parallelism runtime library and are re-used many times for many tasks.

# Use case 1: Session-specific information

**The Setup:**

We are creating a web application which creates a new thread for each user session. Session information is stored in a thread-local variable:

```
struct session_info {
    int                user_id;
    unsigned long long crypt_key[2];
    …
};

thread_local session_info my_session;
```

# Use case 1: Serial code

```
void process(record& r) {
    decrypt(r, my_session.crypt_key);
    …
}

void on_submit()
{
    record shopping_cart, order_history;
    …

    process(shopping_cart);
    process(order_history);
}
```

thread-local lookup

# Use case 1: Parallelized with PPL

```
void process(record& r) {
    decrypt(r, my_session.crypt_key);
    …
}

void on_submit()
{
    record shopping_cart, order_history;
    …
    Concurrency::parallel_invoke(
        [&] { process(shopping_cart); },
        [&] { process(order_history); } );
}
```

Want TLS bound to user thread, not to worker thread

# Use case 1: Parallelized with Cilk

```
void process(record& r) {
    decrypt(r, my_session.crypt_key);
    …
}

void on_submit()
{
    record shopping_cart, order_history;
    …

    cilk_spawn process(shopping_cart);
    process(order_history);
}
```

Want TLS bound to user thread, not to worker thread

# Use Case 2: Dynamic Cache

**The Setup:**

We wish to save computation time in a multithreaded application by caching previously-computed values in a hashed container. Thread-local storage provides a (seemingly) easy way to implement such a cache without having to worry about synchronizing between threads. The occasional redundant computations caused by the lack of a shared cache add an acceptable cost for our data set.

```
thread_local
my_cache_class<int,complex<double>> cache;
```

# Use case 2: Serial code

```
std::complex<double> compute(int arg) {
    cache_iterator i = cache.find(arg);
    if (i != cache.end()) return i->second;
    return cache[i] = some expensive computation;
}
```

thread-local lookup

```
void g(int inputs[SZ], double outputs[SZ]) {
    for (int i = 0; i < SZ; ++i) {
        outputs[i] = compute(inputs[i]);
    }
}
```

# Use case 2: Parallelized with Cilk

```
std::complex<double> compute(int arg) {
    cache_iterator i = cache.find(arg);
    if (i != cache.end()) return i->second;
    return cache[i] = some expensive computation;
}
                  Want TLS bound to worker

void g(int inputs[SZ], double outputs[SZ]) {
    cilk_for (int i = 0; i < SZ; ++i) {
        outputs[i] = compute(inputs[i]);
    }
}
```

# Use case 2: Parallelized with TBB

```
std::complex<double> compute(int arg) {
    cache_iterator i = cache.find(arg);
    if (i != cache.end()) return i->second;
    return cache[i] = some expensive computation;
}
```

Want TLS bound to worker

```
void g(int inputs[SZ], double outputs[SZ]) {
    parallel_for(0, SZ, 1, [&](int i) {
        outputs[i] = compute(inputs[i]);
    }
}
```

# Use case 3: Task-specific variables

**The Setup:**

We start with a function that receives its argument via a global variable (to avoid parameter-proliferation). The value of the global is set in the caller before the call.

```
record g_record;  // Global

void process_record() {
    int id = g_record.id;
    …
}
```

Argument via global variable

# Use case 3: Serial code

Process multiple records in a loop.

```
extern record g_record;  // Global

void g() {
    for (int i = 0; i < num_recs; ++i) {
        init_record(g_record);
        g_record.id = i;
        …
        process_record(); // Process g_record
    }
}
```

# Use case 3: Naïve Parallelization

Process multiple records in a *parallel* loop.

```
extern record g_record;  // Global

void g() {
    cilk_for (int i = 0; i < num_recs; ++i) {
        init_record(g_record);
        g_record.id = i;
        …
        process_record(); // Process g_record
    }
}
```

Each iteration is a separate task

Races!

# Use case 3: Parallelization with TLS

Mitigate races using TLS

```
extern thread_local record g_record;

void g() {
    cilk_for (int i = 0; i < num_recs; ++i) {
        init_record(g_record);
        g_record.id = i;
        …
        process_record(); // Process g_record
    }
}
```

Want per-worker TLS (sort of)

Keep last value of g_record. How do we make other workers destroy their copies?

# Analysis: Comparing use cases 1 & 2

- Use case 1: Session-specific information
  - All parallel tasks share a common user-level TLS object.
  - The *Thread* in *Thread-local* refers to the user-created `std::thread` (or main thread), not to the system-created worker thread.
  - If a task writes to the TLS carelessly, it could cause a race. (Parallelism can always create races if care is not taken.)
- Use case 2: Dynamic Cache
  - Each worker has its own copy of each TLS object.
  - The *Thread* in *Thread-local* refers to the worker thread, not necessarily the user-created `std::thread`.
  - Tasks can still race on TLS, but that would require communicating addresses across tasks.

# Analysis: Use case 3

- Use case 3: Task-specific variables
  - In the parallel code, each worker has its own copy of each TLS object, as in use case 2.
  - In the serial code, only one object should remain, as in use case 1.
  - Ideally, the *T* in *TLS* refers to *Task* rather than *Thread* for the specified variable.
  - The Cilk Plus library provides a *holder* hyperobject (similar to a reducer hyperobject) that implements task-local storage.

# What kind of TLS do we need?

- The three use cases described here show that we need:
  1. Thread-local storage shared by all the tasks executed by a user-created thread.
  2. Worker-local storage that is owned by the system workers and which might survive any given task or user-created thread.
  3. Task-local storage that acts like worker-local storage but is deallocated at the end of a parallel task.

# What should `thread_local` specify?

- The `thread_local` storage class was introduced in C++11 at the same time as `std::thread`.
- The concepts of *thread* and *thread-local* should be consistent.
- Therefore, `thread_local` should specify that the variable is specific to an `std::thread` (as per use case 1).

# Conclusions

- No single concept of thread local storage suffices for all existing and anticipated use cases.
- We will need a *task* formalism and possibly a *worker* formalism in addition to the existing *thread* formalism.
- User-thread-local, worker-local, and task-local storage should all be available via language and/or library features.

# Optimization Notice

**Optimization Notice**

Intel® compilers, associated libraries and associated development tools may include or utilize options that optimize for instruction sets that are available in both Intel® and non-Intel microprocessors (for example SIMD instruction sets), but do not optimize equally for non-Intel microprocessors. In addition, certain compiler options for Intel compilers, including some that are not specific to Intel micro-architecture, are reserved for Intel microprocessors. For a detailed description of Intel compiler options, including the instruction sets and specific microprocessors they implicate, please refer to the “Intel® Compiler User and Reference Guides” under “Compiler Options." Many library routines that are part of Intel® compiler products are more highly optimized for Intel microprocessors than for other microprocessors. While the compilers and libraries in Intel® compiler products offer optimizations for both Intel and Intel-compatible microprocessors, depending on the options you select, your code and other factors, you likely will get extra performance on Intel microprocessors.

Intel® compilers, associated libraries and associated development tools may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include Intel® Streaming SIMD Extensions 2 (Intel® SSE2), Intel® Streaming SIMD Extensions 3 (Intel® SSE3), and Supplemental Streaming SIMD Extensions 3 (Intel® SSSE3) instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors.

While Intel believes our compilers and libraries are excellent choices to assist in obtaining the best performance on Intel® and non-Intel microprocessors, Intel recommends that you evaluate other compilers and libraries to determine which best meet your requirements. We hope to win your business by striving to offer the best performance of any compiler or library; please let us know if you find we do not.

Notice revision #20101101

# Legal Disclaimer



http://intel.com/software/products