Plain Threads are the GOTO of Today’s Computing

Plain Threads Considered Harmful

Hartmut Kaiser (Hartmut.Kaiser@gmail.com)
GOTO Considered Harmful

- Edsger Dijkstra (1968):

Since a number of years I am familiar with the observation that the quality of programmers is a decreasing function of the density of go to statements in the programs they produce. Later I discovered why the use of the go to statement has such disastrous effects and did I become convinced that the go to statement should be abolished from all "higher level"
Plain Threads Considered Harmful

- A large fraction of the flaws in software development are due to programmers not fully understanding all the possible states their code may execute in. In a multithreaded environment, the lack of understanding and the resulting problems are greatly amplified, almost to the point of panic if you are paying attention.

Programming in a functional style makes the state presented to your code explicit, which makes it much easier to reason about, and, in a completely pure system, makes thread race conditions impossible.

John Carmack: In-depth: Functional programming in C++
Hartmut Kaiser: Plain Threads are the GOTO of Today’s Computing

Plain Threads Considered Harmful
What’s a ‘Thread’?

• C++ Standard defines it as follows:

  • **Thread of execution**: “single flow of control within a program” (§1.10p1)
  • **std::thread**: is specified as a component “that can be used to create and manage threads” (see §30.1p1 and §30.3p1), where “threads” explicitly refers to the definition of ”threads of execution” (see §30.1p1)
    • Note: not defined to be an “OS-thread”
  • **Execution agent**: In §30.2.5.1p1, an execution agent is defined as “an entity such as a thread that may perform work in parallel with other execution agents”.

  N4231: Torvald Riegel: Terms and definitions related to threads
What’s a ‘Thread’?

- An entity which exposes 4 properties:
  - A single flow of control (sequence of op-codes)
  - A program counter marking what’s currently being executed
  - An associated execution context (stack, register set, static and dynamic memory, thread local variables, etc.)
  - A state (initialized, pending, suspended, terminated, etc.)
Plain Threads Considered Harmful

For instance, mentioned in Edward E. Lee’s paper (in 2006): ‘The Problem With Threads’, others have talked about this

Main problems:

1. Threads are not composable
   • It’s impossible to tell whether a library function creates threads itself
   • Prone to massive oversubscription

2. Parallelism can’t be ‘disabled’
   • Program logic is closely tied to parallelism even if two threads don’t necessarily run concurrently
   • Difficult to reason about the whole and the parts independently

3. Difficult to ensure balanced load manually
   • Even smallest differences in runtime of parallel tasks influence overall performance
Plain Threads Considered Harmful

Other issues

• No ‘standard’ way of ‘returning’ values from threads
  • But all require explicit synchronization

• Working with threads makes concurrency explicit
  • Difficult to get right with large number of concurrent threads
  • Difficult to reason about programs using threads

• Threads are SLOW

• We need explicit parallelism, well integrated into C++ instead!
  • We need higher level parallelism constructs
  • We need new programming models helping to express parallelism
Hartmut Kaiser: Plain Threads are the GOTO of Today's Computing
Threads are SLOW
Technology Demands new Response

Tianhe-2’s projected theoretical peak performance: 54.9 PetaFLOPs
16,000 nodes, ~3,200,000 computing cores (32,000 Intel Ivy Bridge Xeons, 48,000 Xeon Phi Accelerators)

Hartmut Kaiser: Plain Threads are the GOTO of Today’s Computing
Amdahl’s Law (Strong Scaling)

\[ S = \frac{1}{(1 - P) + \frac{P}{N}} \]

- \( S \): Speedup
- \( P \): Proportion of parallel code
- \( N \): Number of processors

Figure courtesy of Wikipedia (http://en.wikipedia.org/wiki/Amdahl’s_law)
The 4 Horsemen of the Apocalypse: SLOW

- **Starvation**
  - Insufficient concurrent work to maintain high utilization of resources

- **Latencies**
  - Time-distance delay of remote resource access and services

- **Overheads**
  - Work for management of parallel actions and resources on critical path which are not necessary in sequential variant

- **Waiting** for Contention resolution
  - Delays due to lack of availability of oversubscribed shared resources
The 4 Horsemen of the Apocalypse: **SLOW**

- **Starvation**
  - Insufficient concurrent work to maintain high utilization of resources

- **Latencies**
  - Time-distance delay of remote access and services

- **Overheads**
  - Work for management of parallel actions and resources on critical path which are not necessary in sequential variant

- **Waiting for Contention resolution**
  - Delays due to lack of availability of oversubscribed shared resources

**Impose upper bound on both, weak and strong scaling**
Overheads: The Worst of All?

Execution Time over Grain Size
(for different amounts of overheads per thread)

- Sequential Time
- 1µs Overhead
- 100µs Overhead
- 10ms Overhead
Overheads: The Worst of All?

• Even relatively small amounts of work can benefit from being split into smaller tasks
  • Possibly huge amount of ‘threads’
    • In the previous gedankenexperiment we ended up considering up to 10 million threads
    • Best possible scaling is predicted to be reached when using 10000 threads (for 10 seconds worth of work)

• Several problems
  • Impossible to work with that many kernel threads (p-threads)
  • Impossible to reason about this amount of tasks
  • Requires abstraction mechanism
The Challenges

• We need to find a usable way to fully parallelize the applications

• Goals are
  • Defeat The Four Horsemen
  • Provide manageable paradigms and APIs for handling parallelism
  • Expose asynchrony and parallelism to the programmer without exposing concurrency
  • Make data dependencies explicit, hide notion of ‘thread’, ‘communication’, and ‘data distribution’ as much as possible
Stepping Aside

HPX – A General Purpose Runtime System for Applications of Any Scale
HPX – A General Purpose Runtime System

- Solidly based on a theoretical foundation – a well defined, new execution model (ParalleX)

- Exposes an uniform, standards-oriented API for ease of programming parallel and distributed applications.
  - Enables to write fully asynchronous code using hundreds of millions of threads.
  - Provides unified syntax and semantics for local and remote operations.

- Enables writing applications which out-perform and out-scale existing ones
  - A general purpose parallel runtime system for applications of any scale
    - http://stellar-group.org/libraries/hpx
    - https://github.com/STEllAR-GROUP/hpx/

- Is published under Boost license and has an open, active, and thriving developer community.

- Can be used as a platform for research and experimentation
HPX – The API

- As close as possible to C++11/14 standard library, where appropriate, for instance
  - `std::thread`  
  - `std::mutex`  
  - `std::future`  
  - `std::async`  
  - `std::bind`  
  - `std::function`  
  - `std::tuple`  
  - `std::any`  
  - `std::cout`  
  - `std::parallel::for_each`, etc.  
  - `std::parallel::task_region`  
  - `hpx::thread`  
  - `hpx::mutex`  
  - `hpx::future` (including N4107, ‘Concurrency TS’)  
  - `hpx::async` (including N3632)  
  - `hpx::bind`  
  - `hpx::function`  
  - `hpx::tuple`  
  - `hpx::any` (N3508)  
  - `hpx::cout`  
  - `hpx::parallel::for_each` (N4105, ‘Parallelism TS’)  
  - `hpx::parallel::task_region` (N4088)
Thread Overheads

![Graph showing Threads (Tasks) Executed per Second vs Number of Cores for HPX, Qthreads, and TBB.]

Credit: Bryce Adelstein-Lelbach
Explicit Parallelism

Without Explicit Threads
Explicit Parallelism

- C++ needs stronger support for higher level parallelism
  - C++11 has some basic facilities: future, promise, packaged_task, async
  - More is needed

- Several proposals to the Standardization Committee are under consideration
  - Technical Specification: Transactional Memory
  - Technical Specification: Concurrency (note: misnomer)
  - Technical Specification: Parallelism
  - Other smaller proposals: resumable functions, task regions, executors
What is a (the) future

- A future is an object representing a result which has not been calculated yet

- Enables transparent synchronization with producer
- Hides notion of dealing with threads
- Makes asynchrony manageable
- Allows for composition of several asynchronous operations
- (Turns concurrency into parallelism)
What is a (the) Future?

- Many ways to get hold of a future, simplest way is to use (std) async:

```cpp
int universal_answer() { return 42; }

void deep_thought()
{
    future<int> promised_answer = async(&universal_answer);
    // do other things for 7.5 million years
    cout << promised_answer.get() << endl;  // prints 42, eventually
}
```
Concurrency TS (N4107)

- Misnomer: should be called ‘Asynchrony TS’
- Extensions for std::future
  - Means of sequential composition (add continuation to a future)
    - std::experimental::future<>::then()
  - Means of parallel composition
    - std::experimental::when_all(), std::experimental::when_any()
- Helper facilities
  - std::experimental::make_ready_future()
  - std::experimental::make_exceptional_future()
Compositional facilities

• Sequential composition of futures

```cpp
future<string> make_string()
{
    future<int> f1 = async([]() -> int { return 123; });

    future<string> f2 = f1.then(
        [](future<int> f) -> string
        {
            return to_string(f.get());  // here .get() won’t block
        });

    return f2;
}
```
Compositional facilities

- Parallel composition of futures

```cpp
future<int> test_when_all()
{
    shared_future<int> shared_future1 = async([]() { return 125; });
    future<string> future2 = async([]() { return string("hi"); });

    future<tuple<shared_future<int>, future<string>>> all_f =
        when_all(shared_future1, future2);  // also: when_any, etc.

    future<int> result = all_f.then(
        [](future<tuple<shared_future<int>, future<string>>> f) { return do_work(f.get()); });

    return result;
}
```
Dataflow – The New ‘async’ (HPX)

- What if one or more arguments to ‘async’ are futures themselves?
- Normal behavior: pass futures through to function
- Extended behavior: wait for futures to become ready before invoking the function:

```cpp
template <typename F, typename... Arg>
future<typename result_of<F(Args...)>::type>
    dataflow(F&& f, Arg&&... arg);
```

- If ArgN is a future, then the invocation of F will be delayed
- Non-future arguments are passed through
Parallelism TS (N4105)

- Fork-Join parallelism
  - Used for years: OpenMP, CILK, Java concurrency Framework, Task Parallel Library for .NET

Fork/Join Parallelism
Parallel Algorithms

- Mostly, same semantics as sequential algorithms
  - Additional, first argument: execution_policy

  - sequential_execution_policy: seq
  - parallel_execution_policy: par
  - parallel_vector_execution_policy: par_vec

- Special rules related to exception handling

- Entirely fork/join as algorithms return only after all work has been done
  - Performance of those algorithms depends on high quality schedulers
Parallel Algorithms

<table>
<thead>
<tr>
<th>adjacent_difference</th>
<th>adjacent_find</th>
<th>all_of</th>
<th>any_of</th>
</tr>
</thead>
<tbody>
<tr>
<td>copy</td>
<td>copy_if</td>
<td>copy_n</td>
<td>count</td>
</tr>
<tr>
<td>count_if</td>
<td>equal</td>
<td>exclusive_scan</td>
<td>fill</td>
</tr>
<tr>
<td>fill_n</td>
<td>find</td>
<td>find_end</td>
<td>find_first_of</td>
</tr>
<tr>
<td>find_if</td>
<td>find_not</td>
<td>for_each</td>
<td>for_each_n</td>
</tr>
<tr>
<td>generate</td>
<td>generate_n</td>
<td>includes</td>
<td>inclusive_scan</td>
</tr>
<tr>
<td>inner_product</td>
<td>inplace_merge</td>
<td>is_heap</td>
<td>is_heap_until</td>
</tr>
<tr>
<td>is_partitioned</td>
<td>is_sorted</td>
<td>is_sorted_until</td>
<td>lexicographical_compare</td>
</tr>
<tr>
<td>max_element</td>
<td>merge</td>
<td>min_element</td>
<td>minmax_element</td>
</tr>
<tr>
<td>mismatch</td>
<td>move</td>
<td>none_of</td>
<td>nth_element</td>
</tr>
<tr>
<td>partial_sort</td>
<td>partial_sort_copy</td>
<td>partition</td>
<td>partition_copy</td>
</tr>
<tr>
<td>reduce</td>
<td>remove</td>
<td>remove_copy</td>
<td>remove_copy_if</td>
</tr>
<tr>
<td>remove_if</td>
<td>replace</td>
<td>replace_copy</td>
<td>replace_copy_if</td>
</tr>
<tr>
<td>replace_if</td>
<td>reverse</td>
<td>reverse_copy</td>
<td>rotate</td>
</tr>
<tr>
<td>rotate_copy</td>
<td>search</td>
<td>search_n</td>
<td>set_difference</td>
</tr>
<tr>
<td>set_intersection</td>
<td>set_symmetric_difference</td>
<td>set_union</td>
<td>sort</td>
</tr>
<tr>
<td>stable_partition</td>
<td>stable_sort</td>
<td>swap_ranges</td>
<td>transform</td>
</tr>
<tr>
<td>uninitialized_copy</td>
<td>uninitialized_copy_n</td>
<td>uninitialized_fill</td>
<td>uninitialized_fill_n</td>
</tr>
<tr>
<td>unique</td>
<td>unique_copy</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Parallel Algorithms

```cpp
std::vector<int> v = { 1, 2, 3, 4, 5, 6 };  
std::experimental::transform(
    std::experimental::par, begin(v), end(v),
    [](int i) -> int
    {
        return i + 1;
    });
```
Parallel Algorithms (HPX)

- Extensions: Make all algorithms asynchronous, if needed
  - `parallel_task_execution_policy` (asynchronous version of `parallel_execution_policy`), generated with `par(task)`
  - `sequential_task_execution_policy` (asynchronous version of `sequential_execution_policy`), generated with `seq(task)`
  - In both cases the algorithms return a future<>.

- More is needed
  - Make partition results available to programmer
  - Integration with Eric Niebler’s Ranges
Other Proposals: Executors (N4143)

- Executors
  - Originally planned for Concurrency TS, however not included, now N4143
  - Goal: provide alternative for `std::async()`
  - Core API: `void spawn(Func&&)`
    - Any object exposing this function is an executor
    - Defines various executor types
      - `thread_per_task_executor`, `thread_pool_executor`, `system_executor`, etc.
  - Hides the notion of threads by enforcing to think about self-contained tasks
    - Almost functional
    - No races if ‘tasks’ are side effect free and act on value-type arguments only

Hartmut Kaiser: Plain Threads are the GOTO of Today’s Computing
Executors (HPX)

• Added possibility to pass executors to `async()` and `dataflow()`
• Enables control over where (in what context) the function is spawned
  • Could be UI – thread, or dedicated thread used for IO
• Added executors representing different scheduler types
  • FIFO, LIFO schedulers
  • Non-work-stealing schedulers
  • NUMA aware schedulers.
  • Etc.
• User can easily create new executors for finer control
Resumable Functions (D4134)

- Highly scalable (to hundreds of millions of concurrent co-routines)
- Highly efficient (resume and suspend operations comparable in cost to a function call overhead)
- Seamless interaction with existing facilities with no overhead
- Open ended co-routine machinery allowing library designers to develop co-routine libraries exposing various high-level semantics, such as generators, go-routines, tasks and more.
- Usable in environments where exceptions are forbidden or not available
Resumable Functions (D4134)

- Canonical example:

```cpp
std::future<std::ptrdiff_t> tcp_reader(int total)
{
    char buf[64 * 1024];
    std::ptrdiff_t result = 0;
    auto conn = await Tcp::Connect("127.0.0.1", 1337);
    do {
        auto bytesRead = await conn.Read(buf, sizeof(buf));
        total -= bytesRead;
        result += std::count(buf, buf + bytesRead, 'c');
    } while (total > 0);
    return result;
}
```

Hartmut Kaiser: Plain Threads are the GOTO of Today's Computing

Gor Nishanov: await 2.0, CppCon 2014, https://www.youtube.com/watch?v=KUhSjSbINE
Two Examples
Extending Parallel Algorithms

Sean Parent: C++ Seasoning, Going Native 2013
Extending Parallel Algorithms

- New algorithm: gather

```cpp
template <typename BiIter, typename Pred>
pair<BiIter, BiIter> gather(BiIter f, BiIter l, BiIter p, Pred pred)
{
    BiIter it1 = stable_partition(f, p, not1(pred));
    BiIter it2 = stable_partition(p, l, pred);
    return make_pair(it1, it2);
}
```
Extending Parallel Algorithms

- New algorithm: gather_async

```cpp
template <typename BiIter, typename Pred>
future<pair<BiIter, BiIter>> gather_async(BiIter f, BiIter l, BiIter p, Pred pred)
{
    future<BiIter> f1 = parallel::stable_partition(par(task), f, p, not1(pred));
    future<BiIter> f2 = parallel::stable_partition(par(task), p, l, pred);
    return dataflow(
        unwrapped([](BiIter r1, BiIter r2) { return make_pair(r1, r2); }),
        f1, f2);
}
```
Extending Parallel Algorithms (await)

- New algorithm: `gather_async`

```cpp
template <typename BiIter, typename Pred>
future<pair<BiIter, BiIter>> gather_async(BiIter f, BiIter l, BiIter p, Pred pred)
{
    future<BiIter> f1 = parallel::stable_partition(par(task), f, p, not1(pred));
    future<BiIter> f2 = parallel::stable_partition(par(task), p, l, pred);
    return make_pair(await f1, await f2);
}
```
Matrix Transposition

An extended Example
Matrix Transposition

\[ B = A^T \Rightarrow \]

\[ B \quad A^T \]
Matrix Transposition

```cpp
void transpose(std::vector<double>& A, std::vector<double>& B)
{
    #pragma omp parallel for
    for (std::size_t i = 0; i != order; ++i)
        for (std::size_t j = 0; j != order; ++j)
            B[i + order * j] = A[j + order * i];

    int main()
    {
        std::vector<double> A(order * order);
        std::vector<double> B(order * order);

        transpose(A, B);
    }
```
Matrix Transposition

```cpp
// parallel for
std::vector<double> A(order * order);
std::vector<double> B(order * order);
auto range = irange(0, order);
for_each(par, begin(range), end(range),
    [&](std::size_t i)
    {
        for (std::size_t j = 0; j != order; ++j)
        {
            B[i + order * j] = A[j + order * i];
        }
    });
```
Matrix Transposition (distributed)
Matrix Transposition (distributed)

```cpp
std::size_t my_id = hpx::get_locality_id();
std::size_t num_blocks = hpx::get_num_localities();
std::size_t block_order = order / num_blocks;

std::vector<block> A(num_blocks);
std::vector<block> B(num_blocks);
```
Matrix Transposition (distributed)

```cpp
for (std::size_t b = 0; b != num_blocks; ++b) {
  if (b == my_id) {
    A[b] = block(block_order * order);
    B[b] = block(block_order * order);
    hpx::register_id_with_basename("A", A[b], b);
    hpx::register_id_with_basename("B", B[b], b);
  }
  else {
    A[b] = hpx::find_id_from_basename("A", b);
    B[b] = hpx::find_id_from_basename("B", b);
  }
}
```
Matrix Transposition (distributed)
Matrix Transposition (distributed)

```cpp
std::vector<future<void>> results;
auto range = irange(0, num_blocks);
for_each(seq, begin(range), end(range),
    [&](std::size_t phase)
    {
        future<block_data> f1 = A[phase].get_data(my_id, block_size);
        future<block_data> f2 = B[my_id].get_data(phase, block_size);
        results.push_back(hpx::dataflow(unwrapped(transpose), f1, f2));
    });
wait_all(results);
```
Matrix Transposition (await)

```cpp
auto range = irange(0, num_blocks);
for_each(par, begin(range), end(range),
    [&](std::size_t phase)
    {
        future<block_data> f1 = A[phase].get_data(my_id, block_order);
        future<block_data> f2 = B[my_id].get_data(phase, block_order);
        transpose(await f1, await f2);
    });
```

Hartmut Kaiser: Plain Threads are the GOTO of Today's Computing
Matrix Transposition, Results

Credit: Thomas Heller, runs performed on Edison and Babbage (NERSC)
Conclusions

• Multi-core is the new modality and parallelism is here to stay
  • We need higher level abstractions for threading and parallelism
  • Goal should be to make data dependencies explicit
  • Allow reasoning about massively parallel code

• C++11/14 already defines some basic types but more are needed

• Many C++ standardization proposals are being currently discussed
  • Several technical specifications to be released shortly: Parallelism TS, Concurrency TS, and Transactional Memory TS
  • More proposals are in the pipeline

So stop using plain threads after all
Hartmut Kaiser: Plain Threads are the GOTO of Today’s Computing