A Domain-Specific Embedded Language for Programming Parallel Architectures.

Distributed Computing and Applications to Business, Engineering and Science

September 2013.

Jason M^{c}Guiness
& Colin Egan
University of Hertfordshire

dcabes2013@count-zero.ltd.uk
`http://libjmmcg.sf.net/`

Sequence of Presentation.

- A very pragmatic, practical basis to the talk.
- An introduction: why I am here.
- Why do we need & how do we manage multiple threads?
- Propose a DSEL to enable parallelism.
- Describe the grammar, give resultant theorems.
- Examples, and their motivation.
- Discussion.

Introduction.

Why yet another thread-library presentation?

- Because we still find it hard to write multi-threaded programs correctly.
  - According to programming folklore.
- We haven't successfully replaced the von Neumann architecture:
  - Stored program architectures are still prevalent.
    - Companies don't like to change their compilers.
    - People don't like to recompile their programs to run on the latest architectures.
- The memory wall still affects us:
  - The CPU-instruction retirement rate, i.e. rate at which programs require and generate data, exceeds the the memory bandwidth - a by product of Moore's Law.
    - Modern architectures add extra cores to CPUs, in this instance, extra memory buses which feed into those cores.

A Quick Review of Related Threading Models:

- Compiler-based such as Erlang, UPC or HPF.
  - Corollary: companies/people don't like to change their programming language.
- Profusion of library-based solutions such as Posix Threads and OpenMP, Boost.Threads:
  - Don't have to change the language, nor compiler!
  - Suffer from inheritance anomalies & related issue of entangling the thread-safety, thread scheduling and business logic.
  - Each program becomes bespoke, requiring re-testing for threading and business logic issues.
  - Debugging: very hard, an open area of research.
- Intel's TBB or Cilk.
  - Have limited grammars: Cilk - simple data-flow model, TBB - complex, but invasive API.
- The question of how to implement multi-threaded debuggers correctly an open question.
  - Race conditions commonly "disappear" in the debugger...

The DSEL to Assist Parallelism.

Should have the following properties:

- Target *general purpose threading*, defined as scheduling where conditionals or loop-bounds may not be computed at compile-time, nor memoized.
- Support both data-flow and data-parallel constructs succinctly and naturally within the host language.
- Provide guarantees regarding:
  - deadlocks and race-conditions,
  - the algorithmic complexity of any parallel schedule implemented with it.
- Assist in debugging any use of it.
- Example implementation uses C++ as the host language, so more likely to be used in business.

Grammar Overview: Part 1: *thread-pool-type*.

thread-pool-type → `thread_pool` *work-policy size-policy pool-adaptor*

- A `thread_pool` would contain a collection of threads, which may differ from the number of physical cores.

work-policy → `worker_threads_get_work` | `one_thread_distributes`

- The library should implement the classic work-stealing or master-slave work sharing algorithms.

size-policy → `fixed_size` | `tracks_to_max` | `infinite`

- The *size-policy* combined with the *threading-model* could be used to optimize the implementation of the *thread-pool-type*.

pool-adaptor → *joinability api-type threading-model* $\textit{priority-mode}_{\textit{opt}}$ $\textit{comparator}_{\textit{opt}}$

joinability → `joinable` | `nonjoinable`

- The *joinability* type has been provided to allow for optimizations of the *thread-pool-type*.

api-type → `posix_pthreads` | `IBM_cyclops` | `... omitted for brevity`

threading-model → `sequential_mode` | `heavyweight_threading` | `lightweight_threading`

- This specifier provides a coarse representation of the various implementations of threading in the many architectures.

priority-mode → $\texttt{normal\_fifo}_{\textit{def}}$ | `prioritized_queue`

- The `prioritized_queue` would allow specification of whether certain instances of *work-to-be-mutated* could be mutated before other instances according to the specified *comparator*.

comparator → $\texttt{std::less}_{\textit{def}}$

- A binary function-type that would be used to specify a strict weak-ordering on the elements within the `prioritized_queue`.

Grammar Overview: Part 2: other types.

The *thread-pool-type* should define further terminals for programming convenience:

`execution_context`: An opaque type of future that a transfer returns and a proxy to the `result_type` that the mutation creates.

- Access to the instance of the `result_type` implicitly causes the calling thread to wait until the mutation has been completed: a data-flow operation.
- Implementations of `execution_context` must specifically prohibit: aliasing instances of these types, copying instances of these types and assigning instances of these types.

`joinable`: A modifier for transferring *work-to-be-mutated* into an instance of *thread-pool-type*, a data-flow operation.

`nonjoinable`: Another modifier for transferring *work-to-be-mutated* into an instance of *thread-pool-type*, a data-flow operation.

safe-colln → `safe_colln` *collection-type lock-type*

- This adaptor wraps the *collection-type* and *lock-type* in one object; also providing some thread-safe operations upon and access to the underlying collection.

lock-type → `critical_section_lock_type` | `read_write` | `read_decaying_write`

- A `critical_section_lock_type` would be a single-reader, single-writer lock, a simulation of EREW semantics.
- A `read_write` lock would be a multi-readers, single-write lock, a simulation of CREW semantics.
- A `read_decaying_write` lock would be a specialization of a `read_write` lock that also implements atomic transformation of a write-lock into a read-lock.

collection-type: A standard collection such as an STL-style list or vector, etc.

Grammar Overview: Part 3: Rewrite Rules.

Transfer of *work-to-be-mutated* into an instance of *thread-pool-type* has been defined as follows:

transfer-future → *execution-context-result*$_{opt}$ *thread-pool-type transfer-operation*

execution-context-result → `execution_context <<`

- An `execution_context` should be created only via a transfer of *work-to-be-mutated* with the `joinable` modifier into a `thread_pool` defined with the `joinable` *joinability* type.
- It must be an error to transfer work into a `thread_pool` that has been defined using the `nonjoinable` type.
- An `execution_context` should not be creatable without transferring work, so guaranteed to contain an instance of `result_type` of a mutation, implying data-flow like operation.

transfer-operation → *transfer-modifier-operation*$_{opt}$ *transfer-data-operation*

transfer-modifier-operation → `<<` *transfer-modifier*

transfer-modifier → `joinable` | `nonjoinable`

transfer-data-operation → `<<` *transfer-data*

transfer-data → *work-to-be-mutated* | *data-parallel-algorithm*

The *data-parallel-algorithm*s have been defined as follows:

data-parallel-algorithm → `accumulate` | `...  omitted for brevity`

- The style and arguments of the *data-parallel-algorithm*s should be similar to those of the STL. Specifically they should all take a *safe-colln* as an argument to specify the range and functors as specified within the STL.

Properties of the DSEL.

Due to the restricted properties of the execution contexts and the thread pools a few important results arise:

1. The thread schedule created is only an acyclic, directed graph: a tree.
2. From this property we have proved that the schedule generated is ***deadlock and race-condition free***.
3. Moreover in implementing the STL-style algorithms those implementations are efficient, i.e. there are provable bounds on both the execution time and minimum number of processors required to achieve that time.

Initial Theorems (Proofs in the Paper).

1. CFG is a tree:

Theorem

The CFG of any program must be an acyclic directed graph comprising of at least one singly-rooted tree, but may contain multiple singly-rooted, independent, trees.

2. Race-condition Free:

Theorem

The schedule of a CFG satisfying Theorem 1 should be guaranteed to be free of race-conditions.

3. Deadlock Free:

Theorem

The schedule of a CFG satisfying Theorem 1 should be guaranteed to be free of deadlocks.

Final Theorems (Proofs in the Paper).

1. Race-condition and Deadlock Free:

Corollary

The schedule of a CFG satisfying Theorem 1 should be guaranteed to be free of race-conditions and deadlocks

2. Implements Optimal Schedule:

Theorem

The schedule of a CFG satisfying Theorem 1 should be executed with an algorithmic complexity of at least $O(\log(p))$ *and at most* $O(n)$, *in units of time to mutate the work, where n would be the number of work items to be mutated on p processors. The algorithmic order of the minimal time would be poly-logarithmic, so within NC, therefore at least optimal.*

Basic Data-Flow Example.

Listing 1: General-Purpose use of a Thread Pool and Future.

```
struct res_t { int i; };
struct work_type {
   void process(res_t &) {}
};
typedef ppd::thread_pool<
   pool_traits::worker_threads_get_work, pool_traits::fixed_size,
   pool_adaptor<generic_traits::joinable, posix_pthreads, heavyweight_threading>
> pool_type;
typedef pool_type::joinable joinable;
pool_type pool(2);
auto const &context=pool<<joinable()<<work_type();
context->i;
```

- The work has been transferred to the `thread_pool` and the resultant opaque `execution_context` has been captured.
  - `process(res_t &)` is the only invasive artefact of the library for this use-case.
  - The dereference of the proxy conceals the implicit synchronisation:
    - obviously a data-flow operation,
    - an implementation of the *split-phase* constraint.

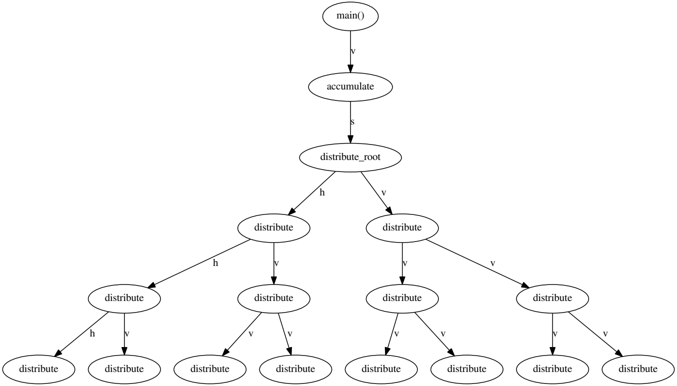
Data-Parallel Example: map-reduce as `accumulate`.

Listing 2: Accumulate with a Thread Pool and Future.

```
typedef ppd::thread_pool<
   pool_traits::worker_threads_get_work, pool_traits::fixed_size,
   pool_adaptor<generic_traits::joinable, posix_pthreads, heavyweight_threading>
> pool_type;
typedef ppd::safe_colln<
   vector<int>, lock_traits::critical_section_lock_type
> vtr_colln_t;
typedef pool_type::joinable joinable;
vtr_colln_t v; v.push_back(1); v.push_back(2);
auto const &context=pool<<joinable()
   <<pool.accumulate(v,1,std::plus<vtr_colln_t::value_type>());
assert(*context==4);
```

- An implementation might:
  - distribute sub-ranges of the *safe-colln*, within the `thread_pool`, performing the mutations sequentially within the sub-ranges, without any locking,
  - compute the final result by combining the intermediate results, the implementation providing suitable locking.
- The *lock-type* of the `safe_colln`:
  - indicates EREW semantics obeyed for access to the collection,
  - released when all of the mutations have completed.

Operation of `accumulate`.



`main()`	the C++ entry-point for the program,
`accumulate` & `distribute_root`	the root-node of the transferred algorithm,
`distribute`	
- internally	distributed the input collection recursively within the graph,
- leaf nodes	performed the mutation upon the sub-range.
`s`	sequential, shown for exposition purposes only,
`v`	vertical, mutation performed by thread within `thread_pool`.
`h`	horizontal, mutation performed by a thread spawned within an `execution_context`. Ensures that sufficient free threads available for `fixed_size thread_pools`.

Discussion.

- A DSEL has been formulated:
  - that targets *general purpose threading* using both data-flow and data-parallel constructs,
  - ensures there should be no deadlocks and race-conditions with guarantees regarding the algorithmic complexity,
  - and assists with debugging any use of it.
- The choice of C++ as a host language was not special.
  - Result should be no surprise: consider the work done relating to auto-parallelizing compilers.
- No need to learn a new programming language, nor change to a novel compiler.
  - Not a panacea: program must be written in a data-flow style.
  - Expose estimate of threading costs.
- Testing the performance with SPEC2006 could be investigated.
  - Perhaps on alternative architectures, GPUs, APUs, etc.