2013 12th International Symposium on Distributed Computing and Applications to Business, Engineering & Science

A Domain-Specific Embedded Language for Programming Parallel Architectures

Jason M^{c}Guiness
Count Zero Limited
London, U.K.
dcabes2013@count-zero.ltd.uk

Colin Egan
CTCA, School of Computer Science
University of Hertfordshire
Hatfield, U.K.

***Abstract*—The authors' goal in this paper has been to define a minimal and orthogonal DSEL (*Domain-Specific Embedded Language*) that would add parallelism to an imperative language. It will be demonstrated that this DSEL will guarantee correct, efficient schedules. The schedules will be shown to be deadlock- and racecondition-free. The efficiency of the schedules will be shown to add no worse than a poly-logarithmic order to the algorithmic run-time of the program on a CREW-PRAM (*Concurrent-Read, Exclusive-Write, Parallel Random-Access Machine*[15]) or EREW-PRAM (*Exclusive-Read EW-PRAM*[15]) computation model. Furthermore the DSEL assists the user with regards to debugging. An implementation of the DSEL in C++ exists.**

***Keywords*—DSEL; grammar; data-flow; data-parallel; library; parallel; concurrent; PRAM**

I. Introduction

The current von Neumann model of super-scalar computer architectures has lead to increased penalties associated with misses in the memory-subsystem, limiting ILP (Instruction-Level Parallelism), also increased design complexity and power consumption. Fetching instructions from different memory banks, i.e. introducing threads, would allow an increase in ILP.

Hence the increasing prevalence of computers with multiple cores, leading to a rise in the available parallelism to the programming community. This parallelism has been exposed via various approaches: ranging from languages e.g. UPC, compilers e.g. HPF or libraries e.g. OpenMP or Cilk. Yet the common folklore in computer science has been that it is hard to program parallel algorithms correctly.

This paper presents an alternative library-based approach to expose this parallelism by defining a minimal and orthogonal DSEL within the host language of the library. The DSEL proposed differs from other approaches because it guarantees correct, efficient schedules with algorithmic run-time guarantees and assists the user with debugging their parallel programs. An implementation of the DSEL in C++ exists: see [9].

II. Related Work

Summarizing [8]; with varying degrees of success the following work has been done to assist in using this parallelism:

- Auto-parallelizing compilers, via the data-flow community [13].
- Language support: such as Erlang [16] or UPC [4].
- Library support: such as POSIX threads (*pthreads*), OpenMP, Intel's TBB [11] or Cilk [7]. Intel's TBB lacks parallel algorithms, nor provided any correctness guarantees. Also the API it has exposed suffers from mixing code relating to the parallel schedule and the business logic, which would make testing more complex.

III. Motivation

The compiler and language based approaches have been the only way to address both correctness or optimization. If programmers were to take advantage of these, they may have to re-implement their programs, a hard change for businesses, limiting the adoption of new languages or novel compilers.

Amongst the criticisms raised regarding the use of libraries [8], [10] such as pthreads or OpenMP have been:

- When used in object-orientated programming, they have suffered from inheritance anomalies [2].
- A related issue has been entangling the thread-safety, thread scheduling and business logic. Each program becomes bespoke, requiring re-testing for threading and business logic issues.
- Debugging such code has been found to be very hard and an open area of research for some time.

Assuming that the language has to be immutable, a DSEL, defined within that language, by a library that supports the following requirements will now be presented.

IV. The DSEL to assist Parallelism

The DSEL 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[1].
- Support both data-flow and data-parallel constructs succinctly and naturally within the host language.
- Provide guarantees regarding deadlocks and race-conditions.
- Provide guarantees regarding the algorithmic complexity of any parallel schedule implemented with it.

[1] A compile or run-time optimisation technique involving a space-time tradeoff. Re-computation of pure functions with the same arguments may be avoided by caching the result.

978-0-7695-5060-2/13 $26.00 © 2013 IEEE
DOI 10.1109/DCABES.2013.22

- Assist in debugging any use of it.
- Use an existing host language would avoid reimplementation, so more likely to be used in business.

First the grammar of the DSEL will be given together with a discussion of the properties. The theoretical results derived from the grammar of the DSEL will follow then finally an example of using the DSEL will be given.

A. Detailed Grammar of the DSEL

C++ was chosen to be the host language because it has been considered hard to parallelize[2] and has the ability to extend its type system. Familiarity with its grammar, defined in Annex A of the ISO C++ Standard [6], would assist the reader.

Clarifications of the notation used:

- Has been based upon that used for context-free grammars.
- $_{\text{opt}}$ means that the keyword is optional.
- $_{\text{def}}$ specifies the default value from the set of values for the keyword.

Initially the terminals, classified as types, then the rewrite rules that comprise the DSEL will be given in the following sections.

1) Types, or terminals: The primary types used within the DSEL should be obtained from the *thread-pool-type*.

1) Thread pools can be composed with various subtypes that could be used to fundamentally affect the implementation and performance of any client software:

   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. This could allow for implementations to virtualize the multiple cores. An implementation may enforce how an instance of a pool should be destroyed, synchronising the threads to ensure they are destroyed and any work in the process of mutation appropriately terminated before the pool would be finally destroyed.

   work-policy→
   `worker_threads_get_work` | `one_thread_distributes`

   - The library should implement the classic work-stealing or master-slave work sharing algorithms. The specific choice could affect any internal queues that would contain unprocessed work. For example a `worker_threads_get_work` queue might be implemented such that the addition and removal of work could be independent.

   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*.
   - `tracks_to_max` would implement some model of the cost of maintaining threads. If threads had low overheads to create & destroy, then an `infinite` size might be a reasonable approximation, conversely opposite characteristics might be better implemented in a `fixed_size` pool.

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

   joinability→
   `joinable` | `nonjoinable`

   - The *joinability* type has been provided to allow for optimizations of the *thread-pool-type*. A `nonjoinable` *thread-pool-type* could be more simply implemented, but also faster in operation.

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

   - `posix_pthreads` would be an implementation of the `heavyweight_threading` pthreads API. `IBM_cyclops` would be an implementation of the `lightweight_threading` IBM BlueGene/C Cyclops [1] API. The *size-policy* type may also interact with this type.

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

   - This specifier provides a coarse representation of the various implementations of threading in the many architectures. For example pthreads would be `heavyweight_threading` whereas Cyclops would be `lightweight_threading`. Separation of the threading model versus the API allows multiple, different, threading APIs on the same platform, for example if there were to be a GPU available, there could be two different threading models within the same program.
   - The `sequential_mode` has been provided to ensure implementations removal all threading aspects within the implementing library, which would ease the burden on the programmer in identifying bugs within their code. If `sequential_mode` were specified then all threading should be removed from the implementing library. All remaining bugs should reside in the user-code, once debugged, could then be parallelized by changing this specifier and re-compiling. Then any further bugs introduced would be due to bugs within the parallel aspects of their code, or the implementing library. We consider this feature of paramount importance: it directly addresses the complex task of debugging parallel software: the algorithm by which the parallelism should be implemented has been separated from the code implementing the mutations on the data.

[2]Chapter 30, "Thread Support Library" in the C++ Standard has not addressed the properties of the DSEL described in this paper.

priority-mode→
`normal_fifo`$_{\text{def}}$ |
`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→
`std::less`$_{\text{def}}$

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

2) Adapted collections should assist in providing thread-safety and specify the memory-access model of the collection:

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. This access should be provided because of the inheritance anomalies described in [2]. This design decision has been chosen for simplicity.
- The adaptor also provides access to both read-lock and write-lock types, which may be the same, but allow the intent of the operations to be clearer.

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. Different architectures could implement this lock more optimally.
- 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.
- The lock must be used to govern the operations on the collection, and not operations on the items contained within the collection.
- The *lock-type* would be used to specify if EREW or CREW operations would be allowed. For example if EREW semantics have been specified then overlapped dereferences of the `execution_context` resultant from *parallel-algorithm*s operating upon the same instance of a *safe-colln* should be strictly ordered by an implementation. Alternatively if CREW semantics were specified then an implementation may allow read-operations upon the same instance of the *safe-colln* to occur in parallel, assuming they were not blocked by a write operation.

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

3) The *thread-pool-type* should define further sub-types (or terminals in the grammatical sense) 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. This would ensure that the properties of the DSEL have been maintained.

`joinable`:
A modifier for transferring *work-to-be-mutated* into an instance of *thread-pool-type*, a data-flow operation. If the *work-to-be-mutated* were transferred using this modifier, then the return result of the transfer must be an `execution_context`. This should be used to obtain the result of the mutation.

`nonjoinable`:
Another modifier for transferring *work-to-be-mutated* into an instance of *thread-pool-type*, a data-flow operation. If the *work-to-be-mutated* were transferred using this modifier, then the return result of the transfer must be nothing. The mutation could occur at some indeterminate time, the result of which could, for example, be detectable by side effects of the mutation within the `result_type` of the *work-to-be-mutated*.

2) Operators or rewrite rules on the thread-pool-type*:* These operations tie together the types and express the restrictions upon the generation of the CFG (*Control Flow Graph*) that may be created.

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

transfer-future→
execution-context-result$_{\text{opt}}$
thread-pool-type transfer-operation
execution-context-result→
`execution_context <<`

- The token sequence "<<" defines the transfer operation, and also has been used in the definition of the *transfer-modifier-operation*, amongst other places.
- 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$_{\text{opt}}$ *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*

3) The Data-Parallel Operations and Algorithms: This section will describe the various parallel algorithms defined within the DSEL.

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

data-parallel-algorithm→
`accumulate` | `...`

- The style and arguments of the *data-parallel-algorithm*s should be similar to those of the STL in [6]. Specifically they should all take a *safe-colln* as an argument to specify the range and functors as specified within the STL. No explicit support has been made for loop-carried dependencies in the functor argument.
- If the algorithms were to be implemented using techniques described in [5] and [3] then the algorithms would be optimal with $O(\log(p))$ complexity in distributing the work to the thread pool optimal algorithmic complexity of $O\left(\frac{n}{p} - 1 + \log(p)\right)$ where n would be the number of items to be computed and p would be the number of threads, ignoring the operation time of the mutations.

B. Properties of the DSEL

In this section some results will be presented that derive from the previous section, the first of which will show that the CFG should be a tree from which the other results will derive.

In all of the theorems presented, we shall assume that the user should refrain from using any other threading-related items or atomic objects other than those defined in section IV-A and that the work they wish to mutate may not be aliased by any other object.

Theorem 1. *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.*

Proof: From the definitions of the `execution_context` IV-A1.3.3, `joinable` IV-A1.3.3, *transfer-future* IV-A2.1.1 & *execution-context-result* IV-A2.1.1 the transfer of *work-to-be-mutated* into the `thread_pool` may be done only once, the result of which returns a single `execution_context`. This would imply that for a node in the CFG, each transfer to the *thread-pool-type* represents a single forward-edge connecting the `execution_context` with the child-node that contains the mutation. Each node may perform none, one or more transfers resulting in none, one or more child-nodes. A node with no children is a leaf-node, containing only a mutation. The back-edge from the mutation to the parent-node would be the edge connecting the result of the mutation with the dereference of the `execution_context`. Back-edges to multiple parent nodes cannot be created, because of definition IV-A1.3.3, so the `execution_context` and the dereference must occur in the same node. Therefore this sub-graph would be acyclic moreover a tree. According to the definitions of *transfer-future* and *execution-context-result* each child-node would either return via the back edge to the parent or generate a further sub-tree, to which the above properties apply. If the entry-point of the program were to be the single thread that runs `main()`, i.e. the single root, this would be the root of the above tree, thus the whole CFG must be a tree. If there were more entry-points, each one can only generate a tree per entry-point, as the `execution_context`s cannot be aliased nor copied between nodes, by definition. ■

According to the above theorem, one may appreciate that a conforming implementation of the DSEL would implement data-flow in software.

Theorem 2. *The schedule of a CFG satisfying Theorem 1 should be guaranteed to be free of race-conditions.*

Proof: A race-condition in the CFG would be represented by a child node with two parent nodes, with forward-edges connecting the parents to the child. Note that the CFG must an acyclic tree according to Theorem 1, then this sub-graph cannot be represented in a tree, so the schedule must be race-condition free. ■

Theorem 3. *The schedule of a CFG satisfying Theorem 1 should be guaranteed to be free of deadlocks.*

Proof: To create a deadlock would require that `execution_context`s C and D had been shared between two threads. i.e. C had been passed from node A to sibling node B, and vice-versa to D. But aliasing `execution_context`s has been explicitly forbidden by definition IV-A1.3.3. ■

Corollary 4. *The schedule of a CFG satisfying Theorem 1 should be guaranteed to be free of race-conditions and deadlocks.*

Proof: Given a CFG for which Theorem 1 held, it must be proven that the Theorems 2 and 3 should not be mutually exclusive. First suppose that a such CFG could exist that satisfied Theorem 3 but not 2. Therefore multiple forward edges from the same `execution_context` would be allowed, but there

must be multiple back-edges, because of Theorem 1, causing Theorem 3 to not hold, a contradiction, therefore such a CFG cannot exist. Therefore any CFG for which Theorem 1 held, must also satisfy both Theorems 2 and 3. ■

Theorem 5. *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.*

Proof: Given a tree with at most n leaf-nodes. It has been proven in [5] that to distribute n items of work onto p processors may be performed with an algorithmic complexity of $O(\log(n))$. The fastest computation time would be if the schedule were a balanced tree, where the computation time would be the depth of the tree, i.e. $O(\log(n))$ in the same units. If the n items of work were to be greater than the p processors, then $O(\log(p)) \leq O(\log(n))$, so the computation time would be slower than $O(\log(p))$. A node may take at most $O\left(\frac{n}{p} - 1 + \log(p)\right)$ computations according to definition IV-A3.1.1, if a *data-parallel-algorithm* were transferred. The slowest computation time would be if the tree were a chain, i.e. $O(n)$ time. In those cases this implies that a conforming implementation should add at most a constant order to the execution time of the schedule. ■

C. Some Example Usage

Two toy examples are given, based upon an example implementation in a library called PPD (*Parallel Pixie Dust*) [9].

The first example shows the simple data-flow usage of the DSEL.

Listing 1. Data-flow example 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 typedef for the *thread-pool-type* would be needed once and could be held in a configuration trait in a header file.

The second example shows how a data-parallel version of the C++ accumulate algorithm might appear.

Listing 2. Example of a parallel version of an STL algorithm.

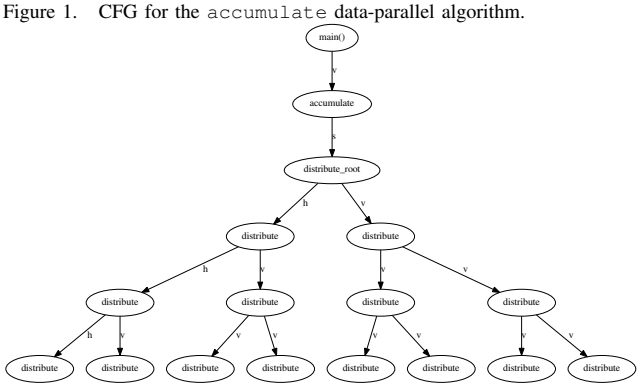
```
typedef ppd::thread_pool<
   pool_traits::worker_threads_get_work, pool_traits::fixed_size,
   pool_adaptor<generic_traits::joinable, posix_pthreads, heavyweight_threading>
> pool_type;
typedef ppd::safe_colln<
   vector<int>, lock_traits::critical_section_lock_type
> vtr_colln_t;
typedef pool_type::joinable 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);
```

This example illustrates a map-reduce operation. An implementation might:

1) take sub-ranges within the *safe-colln,* which would be distributed within the `thread_pool`, within which the mutations would be performed sequentially, their results combined via the functor, without locking any other thread,
2) these results would be combined, the implementation providing suitable locking, computing the final result.

The `accumulate` algorithm produced the CFG in figure 1 ,

Figure 1. CFG for the `accumulate` data-parallel algorithm.



the key is:

- For the node-titles:
  - `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.
- The labels for the edges mean:
  - `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_pool`s.

The input collection has been distributed across eight threads in the `thread_pool` and the CFG generated was a balanced, acyclic tree, satisfying Theorem 1, Corollary 4 and Theorem 5.

V. Conclusions

A DSEL has been formulated:

- targets *general purpose threading* using both data-flow and data-parallel constructs within an existing host language,
- ensures there should be no deadlocks and race-conditions,

- provides guarantees regarding the algorithmic complexity, on a CREW-PRAM or EREW-PRAM computation model, of any schedule implemented
- and assists with debugging any use of it.

Intuition suggests that this result should have come as no surprise considering the work done relating to auto-parallelizing compilers, which work within the AST and CFGs of the parsed program[14].

Note that the DSEL presented in this paper may be hosted in any programming language, the choice of C++ was not special.

Further advantages of this DSEL are that programmers would not need to learn an entirely new programming language, nor would they have to change to a novel compiler implementing the host language, which may not be available, or if it were might be impossible to use for more prosaic business reasons.

VI. Future Work

Investigation of the properties of [9] could be presented, by reimplementing SPEC2006 [12] and contrasting the performance with the literature. The definition of *safe-colln* has not been an optimal design decision: a better approach would have been to define ranges that support locking upon the underlying collection, changing this would not invalidate the rest of the grammar, as this would only affect the overloads to the *data-parallel-algorithm*s. The DSEL may need to be extended to admit memoization.

References

[1] Almasi, G., Cascaval, C., Castanos, J. G., Denneau, M., Lieber, D., Moreira, J. E., and Henry S. Warren, J. Dissecting Cyclops: a detailed analysis of a multithreaded architecture. *SIGARCH Comput. Archit. News 31*, 1 (2003), 26–38.

[2] Bergmans, L. M. *Composing Concurrent Objects*. PhD thesis, Faculty of Electrical Engineering, Mathematics and Computer Science, University of Twente, CopyPrint 2000, Enschede, June 1994.

[3] Casanova, H., Legrand, A., and Robert, Y. *Parallel Algorithms*. Chapman & Hall/CRC Press, 2008.

[4] El-ghazawi, T. A., Carlson, W. W., and Draper, J. M. UPC language specifications v1.1.1. Tech. rep., 2003.

[5] Gibbons, A., and Rytter, W. *Efficient parallel algorithms*. Cambridge University Press, New York, NY, USA, 1988.

[6] ISO. *ISO/IEC 14882:2011 Information technology — Programming languages — C++*. International Organization for Standardization, Geneva, Switzerland, Feb. 2012.

[7] Leiserson, C. E. The Cilk++ concurrency platform. *J. Supercomput. 51*, 3 (Mar. 2010), 244–257.

[8] McGuiness, J. M. Automatic Code-Generation Techniques for Micro-Threaded RISC Architectures. Master's thesis, University of Hertfordshire, Hatfield, Hertfordshire, UK, July 2006.

[9] McGuiness, J. M. libjmmcg - implementing PPD. libjmmcg.sourceforge.net, July 2009.

[10] McGuiness, J. M., Egan, C., Christianson, B., and Gao, G. The Challenges of Efficient Code-Generation for Massively Parallel Architectures. In *Asia-Pacific Computer Systems Architecture Conference* (2006), pp. 416–422.

[11] Pheatt, C. Intel®threading building blocks. *J. Comput. Small Coll. 23*, 4 (2008), 298–298.

[12] Reilly, J. Evolve or Die: Making SPEC's CPU Suite Relevant Today and Tomorrow. In *IISWC* (2006), p. 119.

[13] Snelling, D. F., and Egan, G. K. A Comparative Study of Data-Flow Architectures. Tech. Rep. UMCS-94-4-3, 1994.

[14] Tang, X. *Compiling for Multithreaded Architectures*. PhD thesis, University of Delaware, Delaware, USA, Fall 1999.

[15] Tvrdik, P. Topics in parallel computing - PRAM models. http://pages.cs.wisc.edu/ tvrdik/2/html/Section2.html, January 1999.

[16] Virding, R., Wikström, C., and Williams, M. *Concurrent programming in ERLANG (2nd ed.)*. Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1996.