Disciplined multi-core programming in C

Pjotr Kourzanov¹, Orlando Moreira², and Henk Sips³
¹NXP Research, Eindhoven, Netherlands
²STEricsson Innovation Center, Eindhoven, Netherlands
³Parallel & Distributed Systems group, TU Delft, Netherlands

Abstract

The problem of programmability on modern heterogeneous multi-core and future many-core embedded platforms is still not solved satisfactorily: although many existing but incompatible approaches do provide new languages, language extensions and library interfaces that all focus on specific solutions, and powerful analytical models do exist, no single integrated programming model has been proposed yet for software-defined radio applications or embedded parallel algorithms in general. Our API-less programming model, LIME improves upon this situation by decoupling the functional aspects of a radio from hardware-dependent communication or synchronisation aspects. For the former, we use disciplined programming using standard C and associated language-level constructs, with certain rules and restrictions. For the latter, we introduce a graph-based model specified using a declarative XML schema. We demonstrate a compiler tool-chain for LIME that can parse, verify, analyse and translate radios implemented in this high-level fashion to low-level primitives found in many embedded platforms via platform-specific code generation. Approach of LIME turns out to be extendible to several disciplined models of computation that are important for radio applications, each of which is easily detectable from syntax and structure of radios. We prove that our approach is effective in practice by porting a radio application to LIME and showing a significant decrease in code complexity with no significant increase in run-time overhead due to code generation. Efficiency is the target of our current efforts.

Keywords: SDR, Multi-core, Restricted programming

1. Introduction

It is a well-known fact that modern computing is undergoing radical changes with respect to both the Hardware (HW) architecture and the approach to programming the plentiful of cores that modern HW increasingly contains [1]. Even in the embedded domain, it is not uncommon anymore to find System on Chip (SoC) designs containing multiple, possibly heterogeneous cores connected together by a modern interconnect e.g., a Network on Chip (NoC) or a shared cache, optionally supporting performance guarantees per core/use cases. Many companies and various standardisation bodies are working to ensure the success of this inherently parallel HW template.

The field of embedded Real-Time (RT) Software (SW) includes the sub-domain of radio Base-Band (BB) signal processing SW. Recently, the radio computer has been proposed [10] to address various problems associated to the complexities of future protocols such as the Long Term Evolution (LTE) as well as diversification of protocol standards (3 & 4G wireless, digital radio) and the need to support multiple standards/frequency bands in parallel. A typical Software-Defined Radio (SDR) [11] use-case scenario can exhibit parallelism on many different levels, ranging from embarrassingly parallel as in Multiple-Instruction Multiple-Data (MIMD) with different BBs using different bands running together, Single-Program Multiple-Data (SPMD) with the same BB running using several bands simultaneously to Single-Instruction Multiple-Data (SIMD) with a single BB processing a frame. Also, the adoption of a plethora of new standards requires BB providers to provide some HW independence as to support (even if not in parallel) as many standards as possible. These features of the BB applications make their design, including specification, modelling, implementation, documentation, verification and testing very challenging.

Unlike the parallel HW architectures for embedded applications, which have converged on a fairly unified template with a heterogeneous Multi-Processor SoC (MPSoC) architecture, the problem of SW programmability on heterogeneous MPSoC architectures, however, is still not solved satisfactorily. Many different and incompatible approaches are known (see Section 6).

In Section 2 of this paper, we propose the Less Is More (LIME) Parallel Programming Model (PPM) as an integrated approach which combines (1) disciplined use of C for the algorithmic aspects and (2) a straightforward Extensible Markup Language (XML) schema for communication & synchronisation aspects of SDR applications.
More than one relevant Model of Computation (MoC) can be expressed with LIME programming model, as is further explained in Section 3. Code generation for two very different platforms, the POSIX platform and a Sea-of-DSP (SoD) platform, is further demonstrated in Section 4. Experimental results of using LIME in the design of a Digital Audio Broadcast (DAB) application are reported in Section 5. Finally, we conclude in Section 7.

2. Programming Model

In contrast to prior-art, LIME is neither a library (no specific data-types, explicit functions or primitives), nor an intrusive modification of an old language (no extensions to C are proposed), or a new Turing-complete language.

LIME approach consists of two parts: actor components that contain units of algorithmic work implemented in C, also called limes - algorithms, or actor nodes, and a separate description of the dependencies between these limes. These dependencies are expressed in a dependency graph containing limes as nodes, specified in XML declaratively using the Graph Exchange Format (GXF) schema, or in C using data-structure declarations. The communication and synchronisation logic of the original code, which is usually implemented using library APIs, is transformed to a graph description. This transformation is extremely important for multi-cores since every architecture has specific memory hierarchy and/or a dedicated NoC that inter-connects multiple cores together. Each vendor prefers its own caching strategy and supports different levels of QoS in its interconnect. Besides isolation of these concerns in an abstract graph description, the remaining algorithms benefit from from better instruction and data cache locality.

GXF is an XML schema that has limited expressiveness, i.e. it does not support programming with flow of control/data constructs. An example is shown in the figure above, where the left part depicts the original C code and the right part depicts the limes (top right) and related GXF description (bottom right). Another example can be found in 5.1 where a C data-structure is used to specify an actor containing two limes with an execution order dependency.

After transformation, the resulting limes only contains the algorithm as a step-function, links to the GXF dependency expressions via ports, and uses no platform-specific communication mechanisms directly. As shall be seen in 2.1 the port is represented in C by the function array argument with constant static size, and referred to in XML with a port-id attribute (the port itself is described with a <port> element). The return type of the main() function is void: no information is returned back to the framework dynamically at run-time. This makes it trivial to determine the DF model obeyed by a graph, and possibly apply static scheduling (see Section 3).

LIME defines a set of rules that are enforced by the tool-chain. On one level, the rules restrict the use of C, enforcing disciplined sequential use (see 2.1).

On another level, LIME mandates the use of a declarative GFX schema for the specification of the communication & synchronisation structure. Such structure can in fact contain either Data-Flow (DF), Control-Flow (CF) or Series-Parallel (SP) patterns, in any order. Although the current schema defines only 11 elements (see above), the syntax is easily extendible by introduction of new elements, element values and value-types. The semantics include rules for connectivity, hierarchy, composition and scoping (beyond the scope of this paper).

2.1 Syntax by example

The current syntax builds on existing concepts found in the ANSI C99 standard, as well as on XML with the GFX schema, which has a semi-standard subset 21. The same basic ideas, however, can also be applied to languages other than C and XML. For example, the graphs can be entered in a visual way and saved using the DOT format used in Graphviz 22, while C# could have been used to specify the algorithms. Focusing on SDR in this paper, however, we use C and GFX in some figures while showing an equivalent DOT rendering elsewhere.

We shall give an exposition of LIME using an example from the streaming domain, with a graph that connects (1) source actor with an out-port, (2) copy actor with in-port and an out-port and a (3) sink actor with an in-port. These actors are connected using edges typed “fifo”. Although the exact implementation of communication channels is of course hardware-dependent and subject to specific optimisations (see Section 4 and 5), this does specify the behaviour of ports attached by such edges.

Complete sources for limes actors using the C K&R syntax, together with the graph above provide enough information for our compiler called slimer to generate (1) the actual platform-specific shells, (2) the Operating System (OS) configuration, and (3) startup code to run the streaming graph on an embedded parallel platform. The following syntactic properties of LIME can be directly observed from this simple application:
• no explicit communication or concrete synchronisation calls are present in the input.
• data-dependencies, or ports, are out-ports by default while in-ports are denoted by the C `const` qualifier.
• data-rates are explicit as C array size specifiers.
• the C99 `restrict` keyword can be used to indicate to the compiler that the port’s data is never aliased. See 2.3 for more details.
• in- and out-port rates do not have to match (see figure above), providing a source of Data-Level Parallelism (DLP) to LIME. If a producer writes more data than a consumer can read, we can, under some conditions, choose to create multiple consumers.

2.2 Semantics

There are 3 semantic rules related to the supported models of computation in LIME that focus on the embedding of parallel abstractions in standard C and avoid task management routines related to life-cycle, scheduling, communication, and synchronisation that are inevitably specific to each particular architecture:

1) The C function call as task activation - all inputs are assumed to be ready (e.g., a read-lock) and output space is assumed to be available (e.g., a write-lock).
2) The C function return as task de-activation - all inputs and outputs are flushed (e.g., an unlock) and can not be used by this lime until the next activation.
3) A C function can not by default assume anything about the order of activation.

Exact implementation of instructions that mandate a particular release consistency model (when to do acquires/releases, memory flushes) is completely defined by a particular Back-End (BE) in LIME that is used to compile the application. Because of this decoupling, the BE can choose to apply double-buffering, or in-place processing, depending on platform and/or application requirements. Similarly, the BE may opt to generate non-blocking vs. blocking primitives depending on whether it generates code for a collaborative (see 2.4) vs. preemptive scheduler (see 2.2). Also the classic trade-off between the blocking Synchronous DF (SDF) vs. the non-blocking Kahn Process Network (KPN) write semantic, as well as an optimised implementation of the Parks algorithm 2 for a KPN can be made automatically.

The only invariant that is maintained by LIME for each iteration and each port of an actor is that all input and output data buffers are contiguous and exactly the specified by the data-rate amount of information is directly accessible via a pointer argument, allowing the C toolchain to effectively exploit Instruction-Level Parallelism (ILP) localised in an actor’s lime algorithm.

2.3 Handling statefulness

Any realistic PPM for embedded RT systems has to tackle practical issues associated to multiple instantiation, in addition to the complication of having loop delays as required by SDF analysis techniques. In LIME, both of these are modelled using constructor and destructor limes, which are syntactically not different from regular limes, as can be seen in the figure below. What distinguishes them from other limes is they way they are inter-connected.

This is in fact specified in the dependency graph by subtyping edges as “init” or “deinit”. Above is an example of an actor in which the state port does not use the First-In First-Out (FIFO) protocol but rather maps to memory that is shared between lime instances’ activations. Such ports can not use the restrict semantic since the input and the output port both alias the same (state) memory. An edge specifies the “state” type to indicate to LIME that the ports connected by this edge are state ports, as in the figure below. This information is also needed by the tool to ensure validity and to calculate the state size per actor.

2.4 Compilation flow

A top-level compiler driver slimer sequentially initiates the following compilation engines, some of which (e.g., the ME) may be omitted:

1) Front-End (FE) parsing: converting C algorithms and GFX graphs into machine-readable XML format. Algorithms may also be compiled here, supporting binary component delivery.
2) Middle-End (ME) analysis & scheduling also: static task admission, mapping, grouping, see Section 3
3) BE code generation: of platform-specific shells, OS configuration, and startup code, see Section 5
4) C tool-chain: compiles generated code and (optionally) algorithms, leveraging inlining optimization
5) Profiling & simulation: provides feedback.

3. Support for different MoCs

Various flavours of data-flow modelling allow for different temporal analysis levels, and for static resource allocation with scheduling or buffer minimisation.

The Dynamic DF (DDF) model can express the full range of Turing-complete programs, but lacks many useful
analytical properties. It may be impossible to verify for an arbitrary DDF graph that the synchronisation structure it specifies is deadlock-free. On the other hand, static dataflow models such as SDF [2], Homogeneous Synchronous DF (HSDF) [2], Cyclo-Static DF (CSDF) [3] do allow for temporal analysis. This enables verification of properties such as deadlock-freedom, latency and throughput constraints [12] as well as maximum achievable throughput [7] and minimised FIFO buffers [8] and even the generation of rate-optimal schedules.

It is clear that static models limit applications to ones that work with fixed data rates i.e., amount of data transferred per task activation is not dependent on input data. Thus SDF models tend to be reserved for application domains where RT guarantees are required and where task activation is strictly data-driven. Other variants exist between these two extremes; e.g., Boolean DF (BDF) [6] does not support static scheduling, but does allow generation of quasi-static schedules [3]. A case with a boolean conditional is depicted in [5].

LIME as presented in Section 2 supports the KPN and the SDF models. The syntax and semantics of it, however, can be stretched to support advanced models such as CSDF, BDF and Variable Rate DF (VRDF). To save space, only some of them are discussed here.

3.1 CSDF

This DF variant [5] allows fine-grained decomposition of actors into sub-components each having its own dependencies. CSDF is expressed in LIME simply by specifying several limes in one actor component and defining a local static schedule to order them. As depicted below, the actor signature is the union of all lime signatures. With strict, or eager semantics, CSDF supports encapsulation. With relaxed, or lazy semantics, CSDF additionally supports late-acquire and early-release optimisation schemes. The BE engine can optimise unnoticed acquires and releases as well as in-line code to improve ILP.

3.2 BDF

As the concept of iteration is inherent in LIME (all limes are activated repeatedly until there is no input), only the concept of asynchronous activation and the concept of conditional activation needs extra attention. Both of these CF constructs are supported, the former by VRDF (omitted from this paper) and the latter by variant record port types, containing an enum and a union, see figure below.

Also, non-deterministic CF constructs can be expressed using a similar specification, whereby the switch actor and select actor have only data ports. The activation of actors that connect to such ports becomes purely data-driven. The port for the control tag is left implicit, e.g., as in the figure from Section 5.

4. Tool-flow and code-generation

We have prototyped LIME pragmatically. Rather than implementing our own parser for the C99 language, we have used the following technologies:

- GCC to get the parse-tree dumps.
- AWK scripting to convert signatures and data-types from the dump generated by the GCC to XML.
- AWK for ad-hoc parsing for all XML files.
- SDF ME analyser implemented in OCaml [13, 12].
- AWK for generation of e.g., POSIX code from XML.

This sequence is initiated from slimer, which is built as a shell script that encapsulates a collection of Makefiles and other scripts, which are written carefully to allow parallel compilation. We plan to rewrite these ad-hoc scripting solutions using a single programming/scripting environment, possibly with LIME itself.

4.1 SoD code generation

One baseband platform that is currently being deployed at NXP utilises a heterogeneous MPSoC comprising a number of ARM cores and a number of Xtensa cores, supporting very limited form of shared-memory as well as message-passing primitives via dedicated DMA units to assist data-transfers to and from the host Application Processor (AP). The host is handling higher-level stacks such as the IP and the UI and other control-oriented tasks.

The SoD software that runs on top of this platform contains a lightweight Streaming Kernel (SK) that implements non-preemptive collaborative task scheduling, synchronisation and non-blocking communication primitives (similar to the non-blocking UNIX primitives depicted in [2], as well as a Network Manager (NM) that is used to...
start/stop tasks, configure them, and to setup FIFOs. The SDR tasks and applications are programmed in C directly using proprietary SK and NM APIs, which can be difficult to learn, use and maintain.

As our initial prototype shows, the BE of LIME is able to effectively generate shell wrapper code dealing with kernel primitives as well as all code/data related to the NM setup. The figure above depicts a simplified shell for the copy actor from [21]. Currently, the prototype does not focus on efficiency, and here we show code generation that employs double-buffering. There is no “endless” loop in the generated shell because the scheduler already implements iteration and expects each task to collaborate in the scheduling, hence the return BLOCKED; statements.

4.2 POSIX code generation

```c
#include <stdbool.h>
#include <sys/types.h>
#include <unistd.h>

extern int in[5], out[5];
extern void proc(void *msg, int i[5], int o[5]);

while ((msg = mq_receive(msg_queue, (void *) &msg, &len, 0)) != -1) {
    proc((const char *) msg, i, o);
    mq_send(msg_queue, (void *) &len, sizeof(len), 0);
    returnPopMatrix();
}
```

To test LIME’s proposition of support for platform-independent application design, we have also implemented a POSIX threads (Pthreads) BE which maps application graphs to any Portable Operating System Interface (POSIX) platform using the task management primitives such as `pthread_create()` and POSIX message queue (`mqueue`) calls. We chose to use mqueues because of their simplicity, but of course a different flavour of the same BE could have used pipes, sockets or more efficient POSIX shared memory (shm) and synchronisation primitives such as `pthread_mutex_lock()`.

This platform exhibits properties that are very different from the SoD platform from [1]. A typical Pthreads implementation runs on an homogeneous MPSoC with Symmetric Multi Processing (SMP) and shared-memory, uses a preemptive scheduler and (by default) blocking communication & synchronisation primitives. Because of this, the shell introduces an “endless” loop, within which blocking mqueue primitives are used. A mapping of the same copy actor from [21] is illustrated above.

Despite the differences, the same C99 and GXF files can be processed by the POSIX BE in exactly the same way as by the SoD BE: only the paths (and possibly command-line options) to an appropriate slimer script are different.

5. Experiments with a DAB application

The DAB application was created using proprietary code that has been developed at NXP for demonstration purposes. This includes a library for channel decoding (`libdab`) which is responsible for conversion of the Radio Frequency (RF) signal to a digital data-stream. This includes Orthogonal Frequency Division Multiplexing (OFDM) and Forward Error Correction (FEC) functions. Also, a library for source decoding (`libmad`) that converts the digital data-stream to audio samples is used by this application. This library implements the MPEG-1 Audio Layer 2 (MP2) standard. Modelling DAB using LIME has resulted in the following actors (connected by the graph above): (1) in which gets RF signal from a device/file, (2) chandec which controls the `libdab`, (3) srcdec which controls the `libmad`, (4) out which puts samples to audio device/file, (5) outcons which sets-up the audio device/file, and (6) outdest which finalises the output.

The chandec actor behaves according to the BDF with non-determinate output (control port is implicit). Only valid MP2 slices are sent to the srcdec actor. Other noise or redundancy data is simply discarded; the rate of the port to the hole actor is ZERO, so nothing is sent or received. Note that the same could also have been implemented using the VRDF model of the chandec actor, which would return the length of the slice to the run-time framework of LIME. In this particular case, the BDF and the VRDF models are interchangeable.

As can be seen in above figure, all actors are stateful. For all nodes except out actor, constructors and destructors are internal to the corresponding node. This means that the generated shell includes guarded calls to constructor/destructor `limes` as part of regular processing. This is required when such calls need to run on the core where the regular processing also runs. For out actor this is not necessary because the setup/cleanup of its state can/should be done from the generated `main()`.

Figure below depicts code complexity estimates on left. These were obtained from the DAB application using the `pmccabe` tool. The first column cluster depicts combined results for the in actor, the chandec actor and the srcdec actor, followed by a cluster for each actor in separation. In each cluster, every second column from the left (and denoted by an uppercase letter) relates to results from the baseline: (I/L) uncommented non-blank lines per function, (S/S) statements inside all C functions, traditional (c/C) and modified (c+/C+) McCabe complexity metric.

As can be seen in line 3 of the table below, left-hand side, LIME significantly improves C input code complexity by $\approx 40\%$ over the baseline. The GXF representation of the DAB graph (see graph above) is $\approx 30$ lines of XML, which are formed in a style according to 2. Also, the generated code for both SoD and POSIX platforms does not add extra complexity, as can be seen in lines 6 and 9.

To verify that the code generators of LIME do not introduce unnecessarily large overheads, we ran the DAB application in a cycle, ensuring that the outputs are identical, and compared results with the manually instrumented original baseline code (limes code is instrumented by the
BE automatically of course). The platform used is 32-bit SMP GNU/Linux 2.6.26 4-way multi-core 2 Ghz Intel Architecture (x86) running the SoD user-space simulator and glibc-2.7 (for POSIX code). We have used the GCC 4.3.2 compiler, but enabled no specific optimisations for all software, since our interest is in a relative comparison.

Because we have changed the in actor to do a demand-driven, lazy read from the input file (baseline used eager read of the whole file at startup), there is a considerable increase in system overhead both inside the actor and its shell (Computation resp. Communication in the table below, right-hand side). The srcdec actor has changed slightly: originally it was responsible for writing the output, hence the slight increase in computation & communication latency to the new out actor. This is another reason for omission of the out actor.

Also note that for both baseline and generated SoD scenarios, there is very little overhead in system-level communication, and that for the POSIX-generated scenario, there is very little overhead in user-level communication. This is because the POSIX inqueue functionality is mostly kernel-level, while the SoD simulator uses FIFO mechanism implemented in user-space via shmem.

The DAB application’s latency is dominated by that of the srcdec actor, as can be seen in figure above, right-hand side. We have not experienced any significant changes in the performance of this actor on both platforms.

6. Related work

On one extreme, there are various approaches to build libraries to support the programmer in expressing parallelism, e.g., Phreads, MPI [23] and Mucitcore Communications API (MCAPI) [24]. On the other extreme, there are many new languages proposed that try to embed the constructs of parallel programming in the language itself.

Some of the existing languages such as C++ support extension of the type-system, allowing some flexibility in addressing parallelism in models such as Ct [17] and Threading Building Blocks (TBB) [13]. Other approaches to extend C++ with new keywords include e.g., SPEX [19]. Other languages, most notably C, require intrusive modifications such as introduction of new explicitly parallel constructs, e.g., Parallel C [25], Cilk [26] and others.

We argue that neither specific libraries nor new powerful languages or intrusive modifications of existing languages can present an ultimate embedded industry-wide solution of the programmability problem in the long-term. First, a library Application Programming Interface (API) is always either inevitably tied to a particular HW platform (and efficient), or is standardised (but inevitably inefficient as designed-by-committee), or is either too constrained (but inefficient) or too powerful (but error-prone) for a particular application use-case. Designing a usable library involves making trade-offs that might not be applicable even in the near-term future. Second, adoption of new languages that are not well-known to the embedded community is risky - there is so much proven legacy code written (and experience accumulated) out there using C. Third, usage of solutions that extend C by addition of syntactic sugar (keywords, statements) or #pragma constructs all require expertise that most embedded programmers simply do not have. Correct usage of C++ for sequential code, let alone concurrency, is far from the grasp of the majority.

Given this apparent lack of focus and de-facto standardisation by the industry, one may wonder whether the academic world has a solution. The Ptolemy framework for instance, provides much more than just one MoC, each having very specific and plausible analytical properties, but offers nothing more than just an extensive modelling & simulation environment based on Java. The StreamIt project implements SDF [2] but also struggles to get wide acceptance in the embedded industry mainly because of its usage of a new Java-like programming language albeit extended with DF constructs.

The Y-API (YAPI) library [20] could have served as a focal point to align industrial and academic efforts to obtain a PPM based on C/C++ that supports compositional and predictable programming using several MoCs. However, focusing only on KPN modelling & simulation and having a rigid API, YAPI failed to appeal to most embedded designers.

The lack of an academically accepted way to express parallelism, or a PPM that is usable by the embedded industry on a large-scale, is aggravated by the severe time pressure that forces designers in the industry to take the most powerful mechanism possible (read: libraries) and repeatedly re-invent patterns that parallel community has been brewing for decades. However, it is a well-known fact that unrestricted usage of libraries such as Phthreads can lead to breaking of the compositionality.

7. Conclusions

LIME makes several contributions that improve programmability of SDR on current and future multi-core
embedded platforms. First, we have shown in Section 2 that a number of simple restrictions in the use of C lead to much more modular, analysable and composable code. Disciplined use of a known and universally available sequential language, rather than the unrestricted usage of complex library APIs or intrusive modifications of a language allow application designers to focus on algorithm implementation and documentation, rather than the intricacies of parallel programming on specific systems. It also supports binary 3rd party components, and seamless integration of these and other components.

Second, we made it clear that SDR can only cope with the diversity of multi-core communication and synchronisation solutions, if it separately addresses both computation/algorithm on one level and communication/synchronisation on another. For the former, we propose standard C99. For the latter, we propose a declarative XML schema. Both levels interact using constructs that have well-defined syntax and semantics derived from DF models. These allow analysis with SDF and KPN and support multiple instantation and shared state.

Third, LIME contributes a model that is expressive enough to integrate with several popular MoCs such as the SDF extensions and disciplined CF. As described in Section 3, analytical properties are visible in syntax and structure, allowing our compiler to perform correct-by-construction analysis of the models on the source-level.

Fourth, we have demonstrated a prototype for LIME in Section 4 using the GCC and UNIX scripting. This shows that the proposed model can be easily and effectively re-targeted to various multi-core platforms such as the SoD and standard POSIX. Code generation is effective, though not optimised for run-time efficiency yet.

Finally, a realistic SDR application has been ported to LIME. We have observed a significant ($\approx 40\%$) decrease in DAB code complexity when using LIME, as reported in Section 5. Due to porting effects, run-time performance of some components has decreased insignificantly in the generated communication (shell) code, while that of the whole application has not changed overall. Performance of the major algorithm itself has even improved for the DAB chandec actor. All lime code snippets in this paper are complete and can be compiled using LIME, illustrating that a combination of literate programming and disciplined multi-core programming is helpful during the whole life-cycle of the SDR algorithms. We think that besides productivity and efficiency concerns, perspicuity and traceability are very important contributors to maintainability of embedded software.

We believe LIME to be a promising and a practical approach towards an integrated environment with which the embedded SDR community can make an evolutionary step towards structured system design for multi-cores.

Similarly to the the well-known outcome of the goto vs. structured control-flow statements debate, the direct uses of the assembly of parallel programming, the message-passing (communication) and the shared-memory (synchronisation) primitives, will have to be all considered harmful [15, 16]. History teaches that such low-level constructs should always be available, but well-hidden by a compiler framework behind abstractions that are already accepted in the field of parallel & concurrent programming as simple and straightforward.

References