Programming Languages for Real-Time Applications Executing on Parallel Hardware

Alan Burns

Department of Computer Science, University of York, UK

Panel Position Statement

If there were only one type of parallel hardware then perhaps the problem of designing programming languages for this domain would be tractable. Unfortunately, there are many: multicores, SMPs, MPSoCs, FPGAs, GPGPUs and dataflow machines to name just a few. And even the single architecture of 'multicore' represents a host of alternatives specifically with respect to memory management and scale. The scale issue being a particular source of concern – dual and four cores chips are currently problematic, but we know 1024 cores are not too far away. Even a focus of 1024 cores is sometimes criticized as being redundant and a wasted effort as 10,000 cores per chip is just around the corner.

It is clear that certain forms of parallel hardware are best exploited by tools that extract the necessary concurrency from the application's program which may be sequential or agnostic in this respect. But it is also clear that some forms of hardware and some kinds of application need to let the programmer control the mapping of their code on to the available platform. One of these platforms is SMPs (possibly a multicore chip) and one of the application areas is real-time systems.

Real-time systems have timing constraint, typically deadlines, that must be satisfied. A non real-time system may have considerable non-determinacy; as long as it is making progress, correctness does not depend on the detailed ordering of the large set of possible execution sequences. Real-time system must be more constrained; resources must be managed so that temporal constraints are taken into account. For a relatively small number of cores, the best way of managing this resource usage, at the programming language level, would appear to be the use of the well formed abstraction of a task/process/thread – I'll use the term *task* here.

On single processor system, the execution behaviour of a collection of tasks can be adequately controlled, from a scheduling point of view, by the use of priority and/or explicit task deadlines. What multicore architectures bring, in addition, is the notion of affinity – the relation of a task to the core (or cores) on which it must/may execute. For a real-time system, executing on a multicore platform, control over affinity is as important as the control of priority or deadline.

Most languages used for programming real-time systems provide the abstraction of a task. Sometimes this is well integrated into the language (as with Ada) and sometimes it is more of an add-on only available via a standard library (as with Java). These languages all support the notion of priority and priority based scheduling of tasks, and some even support deadlines and EDF scheduling (Earliest Deadline First). So how should they support affinities?

From the scheduling theoretic point of view, there are a number of task management schemes that all have merit [3]:

- Partitioned allocation – each task is assigned to just one core
- Global allocation – all tasks can execute on all cores
- Migration – some tasks can migrate to some cores
- Zoned allocation – cores are grouped into zones, within which any one of the other schemes may be applied.

So, for an example of the latter scheme, an 80 task application running on a 16 core platform may be structured as four 4-core zones. Within each zone most tasks are partitioned, but three tasks can migrate, each between two statically defined cores [1]. Such a structure has the advantage of giving near optimal performance whilst minimizing overheads – as only 9 tasks migrate, and they do so between just two cores.

The natural place to support these aspects of a task (priority, deadline and affinity) is from within the programming language – even if a different kind of programmer deals with these aspects. It is to be commended that Ada is moving to give this level of support [2].

Turning now to issues of inter-task communication and synchronisation. As the Chair's paper discusses[1], this is a difficult area. What works well on single processors does not even generalise to two cores let alone, 10, 100 or 10,000. At the platform level there are various features that allow fast and predictable parallel code to execute. From the programming language designers point of view, what are the abstractions that allow these features to be accessed? Unlike the uniprocessor situation, it is unlikely that a single scheme will satisfy all application requirements.

One issue that must be addressed is control over the order of execution: when can code be reordered and when must the sequential order in the program be maintained – even when there does not seem to be any functional dependency between the statements. The use of unprotected shared variables is usually deemed undesirable in concurrent programming, but there are a number of lock free schemes that deliver good levels of performance on parallel hardware. These schemes, however, require explicit control over the order of execution of key *volatile* variables. Control over ordering can be extended to define blocks of code as being *atomic*. This allows the compiler and run-time to exploit transactional memory which is becoming more common; although perhaps not yet for real-time systems.

In conclusion, for small numbers of cores the current notion of a sequential task would appear to be the correct abstraction for real-time code. But this simple notion of a task must be augmented by allowing affinity to be controlled, and atomic or volatile variables and blocks to be directly supported in the programming language. There are, however, a number of important issues that are not addressed by this approach:

- Worst-case execution time (WCET), a vital parameter in real-time applications, is not easily obtained/measured on many forms of multicore chips.
- Not all platforms will have homogeneous processors, many will contain various heterogeneous components that will need different forms of abstraction to be available at the language level.

[1] Contained in these proceedings.

– The notion of a task is perhaps not the right abstraction for highly parallel hardware.

Not all of these issues can be solved at the programming language level, but it is to be hoped that the languages available to application developer are more of a help than a hindrance.

References

1. A. Burns, R.I. Davis, P. Wang, and F. Zhang. Partitioned edf scheduling for multiprocessors using a C=D scheme. In *Proceedings of 18th International Conference on Real-Time and Network Systems (RTNS)*, pages 169–178, 2010.
2. A. Burns and A.J. Wellings. Dispatching domains for multiprocessor platforms and their representation in ada. In J. Real and T. Vardanega, editors, *Proceedings of Reliable Software Technologies - Ada-Europe 2010*, volume LNCS 6106, pages 41–53. Springer, 2010.
3. R.I. Davis and A. Burns. A survey of hard real-time scheduling algorithms for multiprocessor systems. *Accepted for publication in ACM Computing Surveys*, 2011.