How to efficiently map apps onto multi-core platforms

By Richard Stahl
Researcher
Smart Systems and Energy Technology Department
Interuniversity Microelectronics Centre

The first multi-core platforms have found their way into embedded systems for entertainment and communication, thanks to their greater computational power, flexibility and energy efficiency. However, this article shows, mapping applications onto these systems remains a challenge that is costly, slow and error-prone.

Although the multi-core programmable architectures have a huge potential to tackle present and future applications, a key issue is still open: How can developers map an application onto such a multi-core platform fast and efficiently, while profiting from the potential benefits of parallel processing?

This question can be reformulated as: What programming model should they use? In a broad sense, a programming model is a set of software technologies and abstractions that provides the designer with means to express the algorithm in a way that matches the target architecture. These software technologies exist at different levels of abstraction and encompass programming languages, libraries, compilers, runtime mapping components etc.

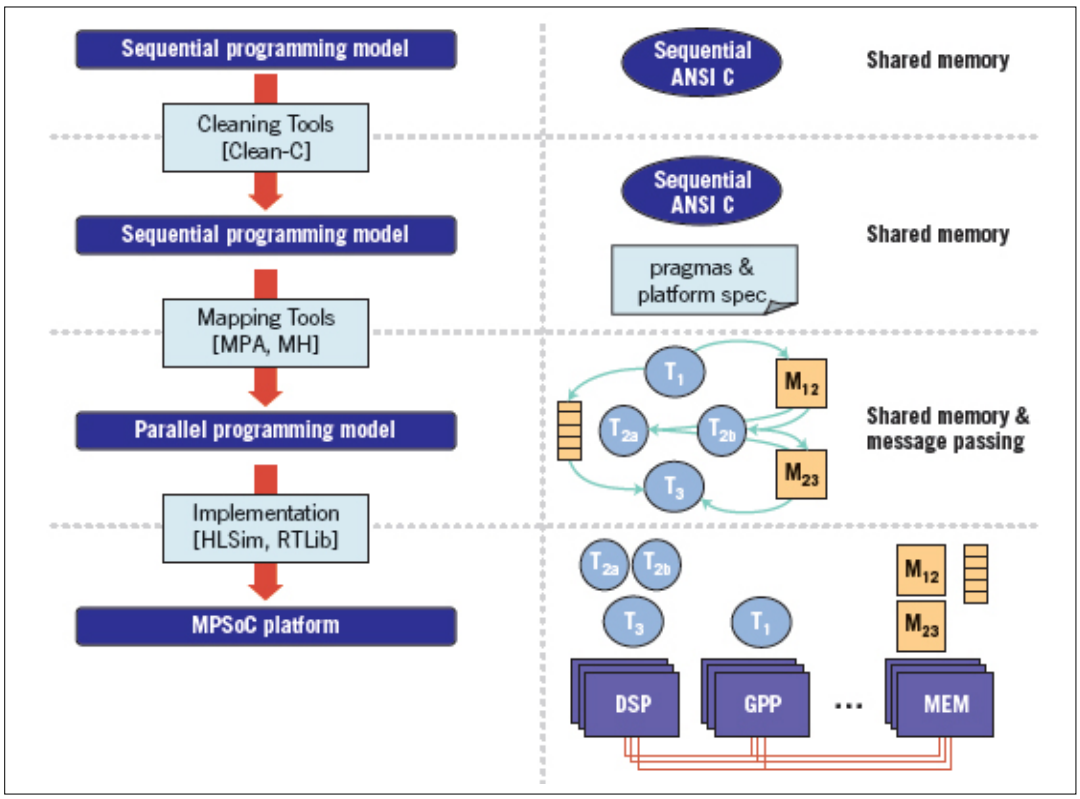

Figure 1: Mapping tools enable user-controlled transformation of sequential C into parallel C (MPA), while optimising data communication among parallel tasks (MH).

Programming model

Obviously, programming a multi-core system requires a sort of parallel programming model. A number of parallel programming languages (OCCAM, HPF, MapReduce), libraries (pThreads, MPI, OpenMP) and other software solutions (auto-parallelizing compilers, e.g. SUIF, Paraphrase-II, Paradigm, Compaan) have been researched in the last decades, yet their main problem has been either their low acceptance among developers or their poor performance.

How are multi-core systems programmed today? Embedded software developers still prefer C/C++ programming. If absolutely necessary, they augment the sequential programs with a parallel library and/or optimised parallel components to achieve the required performance. This seems to work for now, but as the number of cores grows beyond (let's say) 10, the sequential programming model will likely become a considerable burden. Being embedded software developers, we often dream of an auto-parallelizing compiler that analyses sequential C/C++ and/or Java programs, takes into account the target architecture, and, given the application performance requirements, maps the program onto the multi-core platform. Unfortunately, such a super-compiler doesn't exist and chances are rather low that it ever will, especially due to the vastness and complexity of the design space.

How could/should multi-cores be programmed in the future? Dividing the parallel programming challenge into a number of sub-challenges is definitely part of the answer. Current solutions (discussed here in more detail) rely on providing the developers with solid software-tool support (OpenMP, MPA). In the many-core future, the true parallel programming (and thinking) will still be hard, so the developers will most probably like to keep thinking sequentially, while creating highly parallel programs (Maestro/Axum).

Contemporary approaches

Many solutions to parallel programming have been proposed over the last three (or four) decades, most of them originating in high-performance computing. They are either based on the use of a parallel programming model (fully manual or semi-automatic) or they use a fully automatic mapping. The most typical representatives are pThreads, MPI, process networks, transactional memories, data-parallel languages, parallel functional languages, auto-parallelizing compilers and OpenMP. Solutions that are potentially interesting (and in some cases already used) for embedded systems are the following:

MPI is a low-level communication library, typically used in scientific applications on distributed memory machines. MPI is a fully manual, low-level API for distributed systems with clearly defined semantics. In practice, this entails lots of work for the user (such as

deadlocks and correctness), but eventually it brings great results.

Process networks are, in general, a relatively low-level approach, based on a strict formalism (data-flow, Kahn process networks), which allows a better program analysis. Concrete implementations (YAPPI, StreamIt, Ptolemy) have much in common with MPI, while the emphasis is on analysability and a certain level of hardware virtualisation. While highly suited for (specific) embedded applications, their acceptance among embedded systems developers is rather low.

Auto-parallelizing compilers are the holy grail of parallel computing. A number of research (SUIF, Paradigm, Paraphrase-II) and commercial (IBM X1) compilers have been developed over the last three decades. They've eventually achieved some success in automated parallelisation of loop nests, yet almost none in high-level functional parallelism.

OpenMP can be seen as a meet-half-way solution to the challenges of auto-parallelisation. It was originally developed for parallelisation of loops in scientific applications. While the user still needs to express the parallelism explicitly in a form of pragmas in the source code, the underlying compiler and runtime system take care of the rest (data distribution, communication, synchronisation). The user, however, is fully responsible for the correctness of the pragmas and of the resulting parallel program. In the embedded systems domain, in particular, OpenMP has other drawbacks: higher requirements on the run-time system, a shared memory model and the fact that the user has (practically) no control over the final partitioning and resource allocation.

ARM SoC-C is another relatively new and interesting solution de-

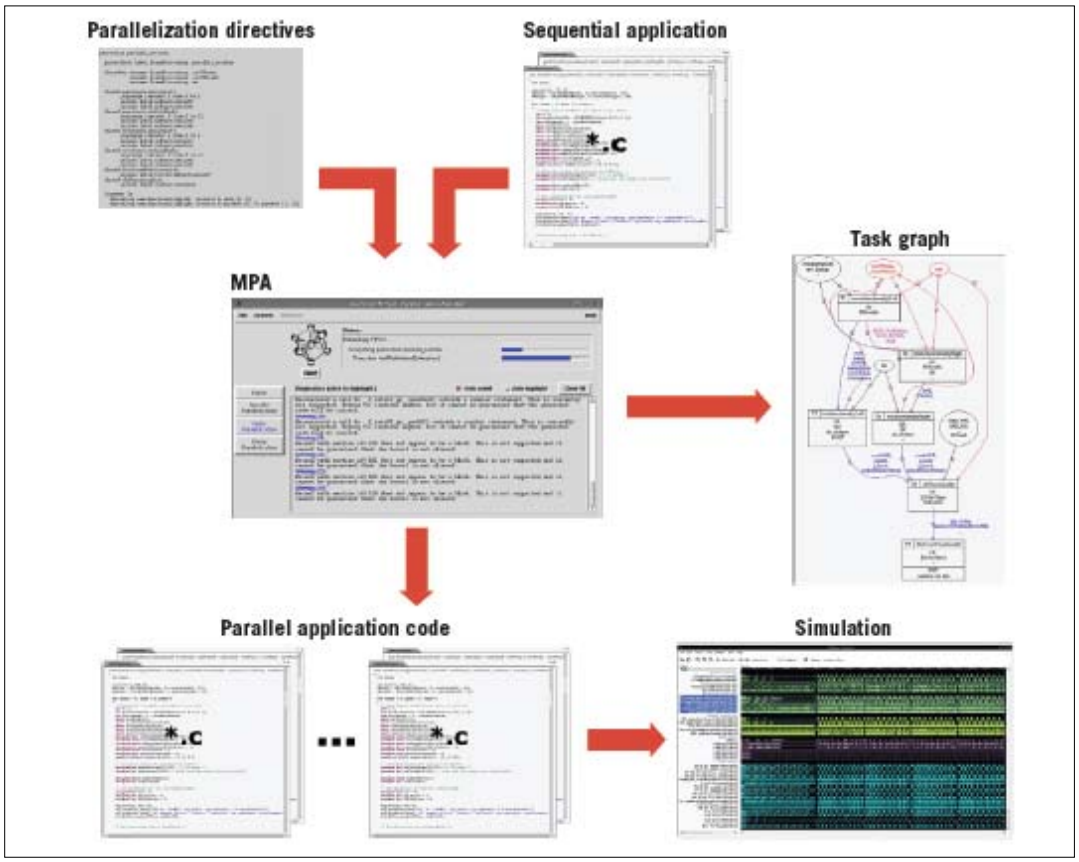

Figure 2: Based on user specified parallelisation directives, MPA performs automated parallelisation of sequential C program into parallel C program.

OpenMP	MPA
(–) Limited support for functional parallelism	(+) Strong support for functional parallelism and pipelines
(–) No real support for loops with loop-carried dependencies	(+) Fully automatic support for loop-carried dependencies
(–) Correctness of parallelism directives is responsibility of the user	(+) Correct-by construction! The tool automatically inserts communication to handle dependencies
(–) Communication via shared memory, potentially resulting in scalability issues	(+) Automatically inserted point-to-point communication channels, with optional shared memory
(–) Assumes homogeneous processors	(+) Usable also for heterogeneous multiprocessor systems
(–) Parallelism pragmas put directly into the source code complicate the design-space exploration	(+) The parallelism specification is kept in a separate file to support easy exploration
(–) Run-time load balancing requires more hardware support, while limiting the user's control over the final partitioning	(+) Light-weight run-time, with user in full control over the partitioning
(–) Reductions identified manually	(+) Reductions identified automatically
(+) No restrictions on host language	(–) Minor restrictions on C
(+) De-facto standard, supported on many platforms	(–) Proprietary solution from IMEC

Table 1: Shown is a comparison of OpenMP and MPA.

veloped specifically for embedded systems. It has a very explicit programming model with an explicit parallelism specification, communication, synchronisation, and explicit data distribution. This, on one hand, means that users have full control over the partitioning process, while, on the other hand, they are also fully responsible for the correctness of the parallel program. Similarly to MPI, any form of parallelism can be realised as the underlying programming model is rather low-level.

As we can see, there is, so to speak, no golden bullet, but rather lots of dedicated solutions to particular problems in specific domains. Moreover, it's still unclear at this point which of these solutions are really suitable for multi-core embedded programming. There is, however, yet another solution at hand: IMEC's Multi-processor Parallelisation Assist (MPA).

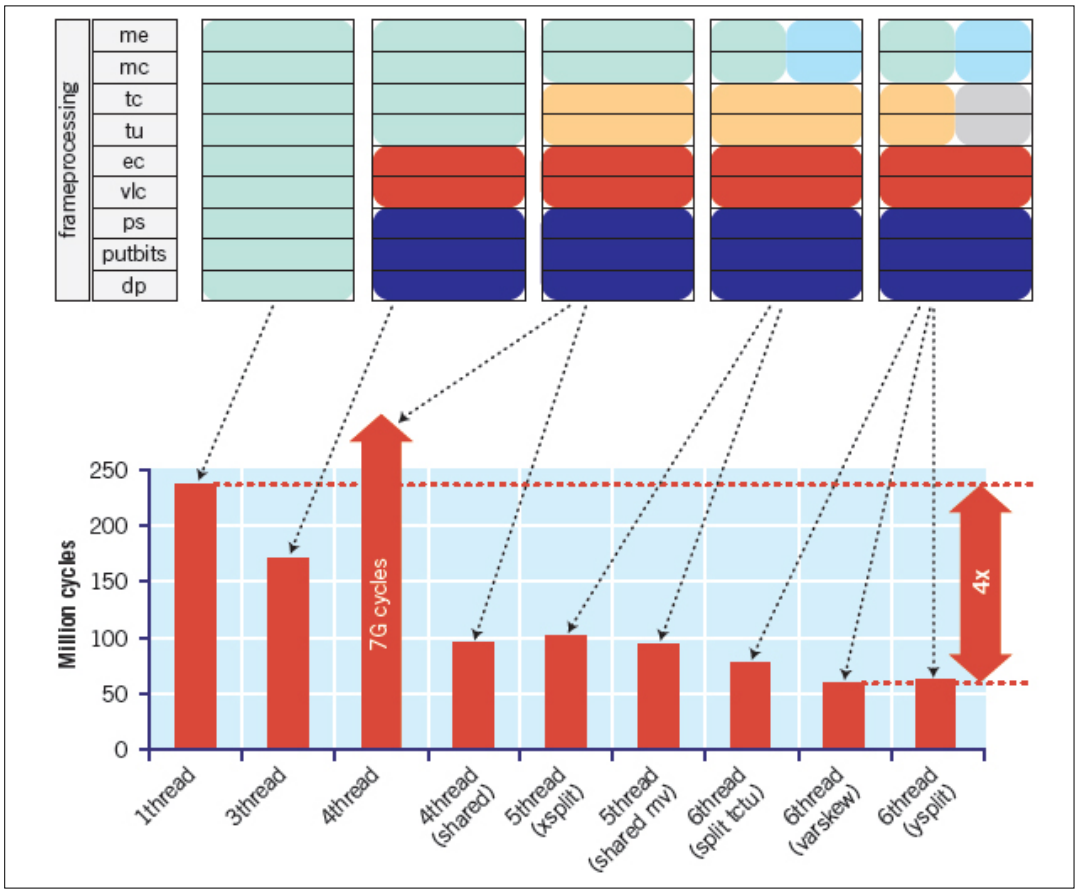

Figure 3: After cleaning the application source code (of approximately 8,600 lines), it took one person one day to generate and evaluate eight parallelised code alternatives.

MPSoC solution

Over the course of many years, the Interuniversity Microelectronics Centre (IMEC) has developed a suite of tools and methods to parallelise applications onto embedded multi-core platforms, with focus on design-time mapping, parallelisation and easy design space exploration. The tool suite comprises MPA, Memory Hierarchy optimisations (MH) and a set of supporting tools for code cleaning (CleanC) and performance estimation (HLSim), as shown in **Figure 1**.

Tech overview

The core of the MPSoC solution is MPA, an interactive parallelisation tool based on IMEC's compiler technology. It automatically transforms sequential C programs into parallel C programs (a process shown in **Figure 2**). Based on user specification, the tool creates functional pipelines, data-split parallel loops, or nested combinations of the two. After analysing the application code, MPA automatically inserts optimised FIFO communication channels to handle inter-task communication and generates parallel C code. The generated code is correct-by-construction and functionally equivalent to the original sequential program, which also means it is deadlock and race-condition free.

MPA enables straightforward, automated splitting of sequential C programs into multiple parallel tasks, while leaving the user in full control of the parallelisation process. This way it enables us to perform fast design-space exploration leading to more energy-efficient multi-core systems and shorter time-to-market.

Comparison with OpenMP

In comparison with OpenMP, MPSoC and MPA technologies have been developed specifically for optimisation of multi-core embedded systems. In the embedded domain in particular, the main focus is not only on increasing the performance, but also on finding trade-offs between performance, energy and area.

The most interesting features of MPA, which enable such optimisations, may be summarised as follows:

- Strong support for functional parallelism, functional pipelining, loop parallelism and their nested combinations, giving the user full freedom in exploration of diverse parallelisation alternatives;
- Explicit specification of computation partitioning, provided by the user in a separate par.spec file(s) in a form of loop ranges and parallel sections. No changes to the source code are required;
- Implicit specification of communication, synchronisation, and data distribution, created automatically by MPA. Detailed information is provided to the user in a form of task-graph, generated parallel source code, and simulation results (HLSim);
- Use of a high-level programming model, similar to process networks that's easy to analyse and understand;
- Automatic dependence analysis and insertion of communication and synchronisation, freeing the user of this complex and error-prone part of the parallelisation process;
- Point-to-point communication, with no shared memory assumed.

A more detailed feature comparison between OpenMP and MPA is provided in **Table 1**.

Case studies

MPA has proven its potential in a number of multimedia and wireless projects at IMEC, as well as in commercial applications of our industrial partners (Samsung and Toshiba).

As an experiment, we parallelised an MPEG-4 encoder to prove that fast design-space exploration using MPA is possible. After cleaning the application source code (of approximately 8,600 lines), it took one person one day to generate and evaluate eight parallelised code alternatives (**Figure 3**). In such a short time, he was able

to speed up the encoder by a factor 4 using six parallel threads in a "6thread (varskew)" alternative.

MPA has also been successfully used at Samsung to map an MPEG-4 encoder onto a 4-core MP-ARM platform. A speed-up factor of 2.5 was achieved with four threads running on four cores.

At IMEC, a wireless application, with 40MHz MIMO SDM-OFDM base band processing, has been parallelised using MPA support for symbolic loop splits. Our developers have reached the real-time performance requirements by processing odd and even symbols in parallel. This way, the symbol processing is speeded up from 5.8µs running on single ADRES processor, to 4.0µs running on two ADRES processors on our dual-core BEAR platform.

The future is dynamic

More and more applications require multiple devices to co-operate to get their work done, such as in an artificial pancreas for diabetic patients that requires multiple networked sensors and actuators spread all over the body. This results in inherently dynamic systems, where the dynamicity is due to hardware (many kinds of devices need to be supported) and software (user interaction, context-specific behaviour of devices and complex and various data formats).

Our vision sees runtime resource management as the key to address the increasing dynamism in software for distributed embedded devices. Runtime resource managers are required at the level of the device and processor, but also at the level of networked devices. They are governed by strategies that are application-dependent and guide how the runtime manager reacts to runtime changes in context, such as the battery running low or reading a low sugar level for a diabetes patient.

Extending our broad knowledge of the multi-core programming solutions upwards to the device and network level will result in runtime resource managers ready to support the next generation of applications for networked nomadic devices.