A unified multicore programming model

Simplifying multicore migration

By Sven Brehmer

Abstract

There are a number of different multicore architectures and programming models available, making it challenging for developers to choose which one to use when migrating applications to multicore. We will look at some of the drivers for and challenges with multicore, discuss the importance of multicore programming models and how to take advantage of multicore standards and rapid prototyping to ease application multicore migrations for ARM Coretex-A9 MPCore.

PolyCore Software, Inc. http://www.polycoresoftware.com	Micrium, Inc. http://www.micrium.com

Table of Contents

1. Multicore Drivers and Challenges

The two most common reasons for considering migrating to multicore are performance, power consumption and consolidation. The demand for more features, functionality and richer user experiences are driving the demand for higher bandwidth data streams and increasing application complexity. To address those user demands, more processing power is needed within a reduced power consumption envelope. The answer is multicore.

1.1. Multicore Drivers

1.1.1. Performance

Through concurrency multicore provides more potential processing power enabling applications to increase functionality, capabilities and therefore leading to complexity. Multicore also offers higher throughput and lower latencies supporting the ever increasing demand for streaming data.

1.1.2. Power Consumption

Multiple cores can offer the same or higher performance than a single processor at lower clock frequency. Power consumption determines the longevity of a battery charge for mobile devices, cooling requirements and therefore mean time between failure (MTBF) for infrastructure devices and energy costs for all devices.

1.1.3. Consolidation

Systems with multiple discrete processors can potentially be consolidated into a single or fewer multicore chip, thus reducing size and the bill of material and cost.

Performance and power consumption characteristics are therefore key factors in our continued ability to provide competitive solutions to end users, and reducing cost. However multicore platforms/devices can only take advantage of these benefits provided that there is sufficient opportunity for concurrency in the application.

1.2. Multicore Challenges

1.2.1. Software Infrastructure

The software infrastructure, both run-time software and development tools have been developed for a single processor.

Programming languages, including C and C++, the most commonly used programming languages in embedded systems, are almost exclusively sequential. There are a few language extensions originating from High Performance Computing (HPC) such as OpenMP[1], primarily targeting shared memory systems and MPI[2] for widely distributed computing.

Besides SMP operating systems and hypervisors, for multicore platforms with homogenous cores and shared memory, there is little in the way run-time multicore support. Because the SMP model is based on shared memory, scalability will be a challenge with an increasing number of cores per chip, competing for access to the memory.

As with run-time, besides tools focusing on SMP environment, not much multicore "relief" is found in the tools department. There is an emergence of tools addressing concurrency, but little in the way of standards to "guarantee" their longevity.

1.2.2. Standards

Industry organizations such as for example the Multicore Association[3] are providing open standards for multicore, with MCAPI[4] 1.0 released two years ago and MRAPI[5] 1.0 and MCAPI version 2.0 to be released before the end of 2010. Even so multicore standards are still in the emerging stage. Multiple areas of multicore will benefit from standardization and some coherency between standardization in those different areas is desirable. MCAPI and MRAPI for example share some concepts, which makes it easier if you know one of them to understand the other and to use them together.

1.2.3. The Human Element

Another aspect of dealing with multicore is the concurrency, which makes it more difficult to understand, we tend to think of and solve problems in a step wise fashion. It's also challenging for tools to provide easy to understand information to the developer about many concurrent events.

2. Programming Models

2.1. Models

An application can be composed in many different ways and several different programming models can be employed with multicore. Some of the models are:

- Task parallelism, where a stream of operations or functions can be organized into tasks that can run concurrently on multiple cores. The efficiency of this model will depend on how independent the operations of the tasks are.
- Data parallelism, where the data being operated on can be divided into chucks that can be processed concurrently on different cores. The efficiency of this model will depend on how independent the chunks of data are.
- SMP[6] - symmetric multiprocessing, uses a single operating system on multiple cores. Processes, tasks or threads are scheduled to run on the different cores as determined by the scheduler. This model requires homogenous cores and shared memory. In some cases the processes/tasks/threads are "pinned" to cores, for various reasons such as for example, a function requires exclusive access to a core to meet performance requirements or a core is dedicated to a legacy application. This is sometimes referred to as bound multiprocessing (BMP), as it is not symmetric any more.
- AMP[7] - asymmetric multiprocessing, may use multiple instantiations of the same or different operating systems, or no OS or a combination of OS and no OS. Processes, tasks or threads are generally assigned specific cores and scheduled to run as determined by the scheduler of the OS on that core. This model works with homogenous and heterogeneous cores and shared memory or non-uniform memory architectures (NUMA).

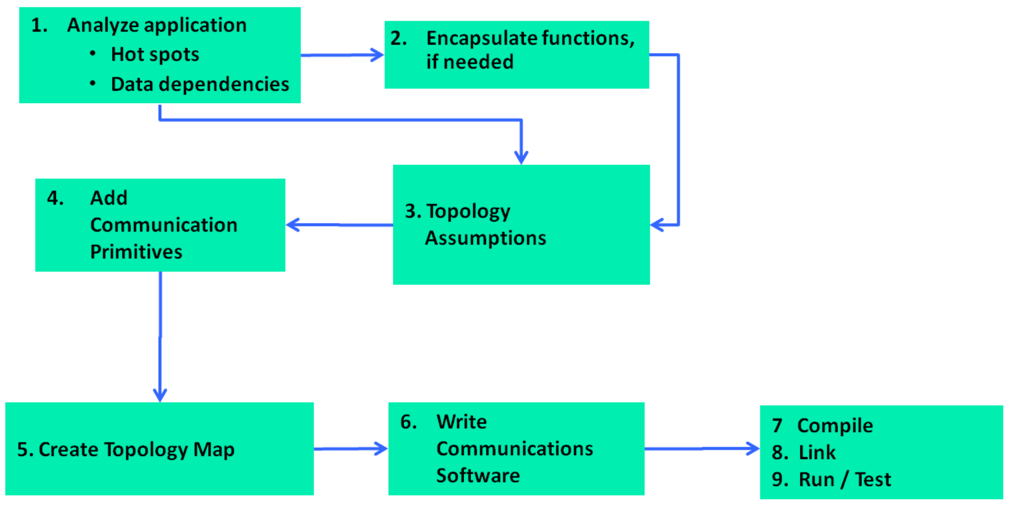
2.2. Which one should I use?

The application characteristics and behavior should drive the underlying multicore platform. A modular application makes it easier to migrate to multicore regardless of the chosen model, and the hardware may or may not already be specified. A consistent programming model that works for both SMP and AMP is a plus. It simplifies migration and future development (code reuse), allowing the application to be deployed in either or a combination of both models in the same system (combined models will be increasingly common as the core count goes up) . A standardized unified model broadens the available ecosystem, both run-time software and development tools, and better preserves software across product lines and for future product generations.

3. Example Application and Platform

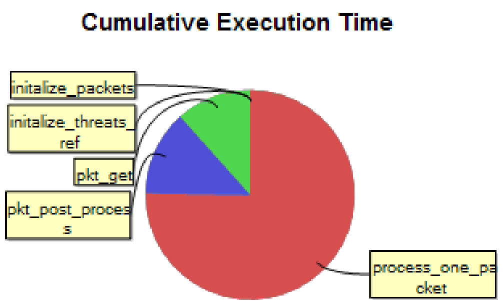
We are going to take look at a specific application and multicore platform to exemplify the application migration process and flow. It is a packet processing application running on a single core ARM Cortex-A9, controlled by an RTOS. The next generation product must provide more packet processing functionality, more advanced configurability and management capabilities and substantially higher throughput. The hardware platform is only partially defined, it will include one or more ARM Cortex-A9 MPCores, and potentially DSP's and FPGA's. Simulators will be used to allow modeling of the platform, to determine that the requirements can be met, before committing to the hardware platform.

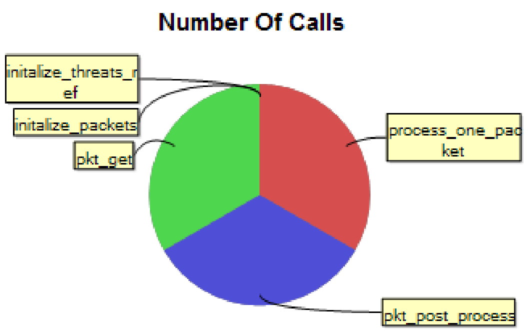
3.1. Application Migration Flow



3.1.1. Analyze

The first step is to find the hot spots. If a function that uses 50% of the CPU cycles is sped up by a factor of 2 we have gained 25%, whereas if a function that takes 10% of the CPU cycles is sped up by a factor of 5, we have gained 8%. With a proper and tools supported method the "higher hanging fruit" can be addressed later if so required.





3.2. Improve Performance and Capabilities With Multicore

As a first step, we try one ARM Cortex-A9 MPCore, with 4 cores, running SMP Linux on 2 of the cores (configuration, management, NW connectivity) and separate instantiations on the RTOS on the other cores.

We find that this configuration is not sufficient to meet our requirements and we add one ARM (quad) MPCore and two 3-core DSP's. The RTOS is used on the additional ARM cores and a DSP OS is used for the DSP cores.

We again find that we need more performance and decide to add two FPGA's that include ARM Cortex-A9 MPCore (dual). Although not needed immediately the FPGA soft cores must also be supported by our programming model, for future expansion. The RTOS will control the soft cores.

In order to make this work we need to effectively communicate between the cores. This communication includes:

- Multiple on-chip buses
- Multiple chip-interconnects
- Multiple types of cores
- Multiple OS'es

3.3. We Need a Communications Model

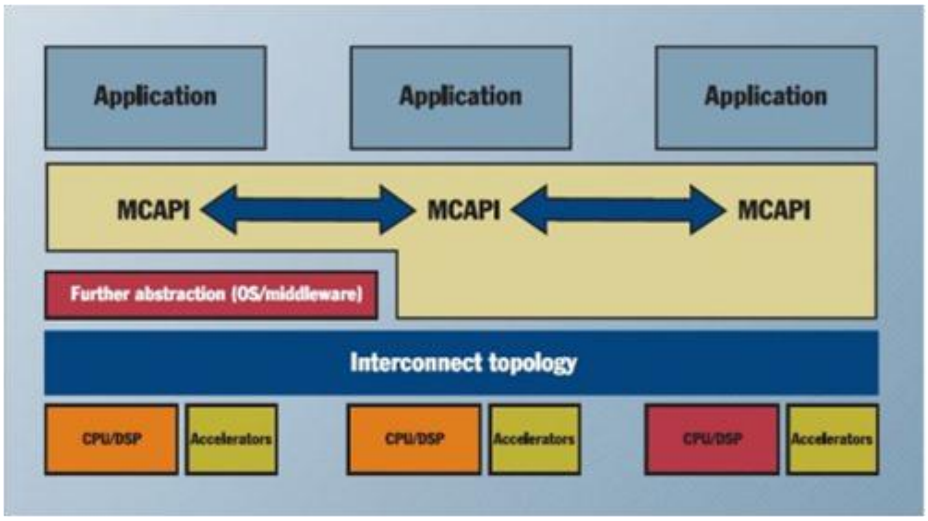
To handle this multitude of hardware and software we need to have a communications model that is scalable across a number of different cores and can provide a unified communications API across different OS'es and transports.

Explicit communication simplifies synchronization across the different entities in the platform. Because the platform includes both on-chip inter-core shared memory and NUMA inter-chip transports, both copy by reference and data movement will be used (on and off chip).

We also need a long term applicable communications model that can be used in future generation products.

3.4. Multicore Standards

MCAPI a standard communications API defined specifically for multicore communications will be used. MCAPI is a source level API that allows implementations to abstract the application from the multitudes of cores, transports and OS'es as discussed above.



MCAPI provides:

- Messages, Packet & Scalar Channels.
- Management Functions
- Basic Topology Discovery
- Standardized Programming Model

MCAPI is defined by a cross industry Multicore Association working group, and is today supported by multiple vendors.

3.4.1. MCAPI Communication Modes

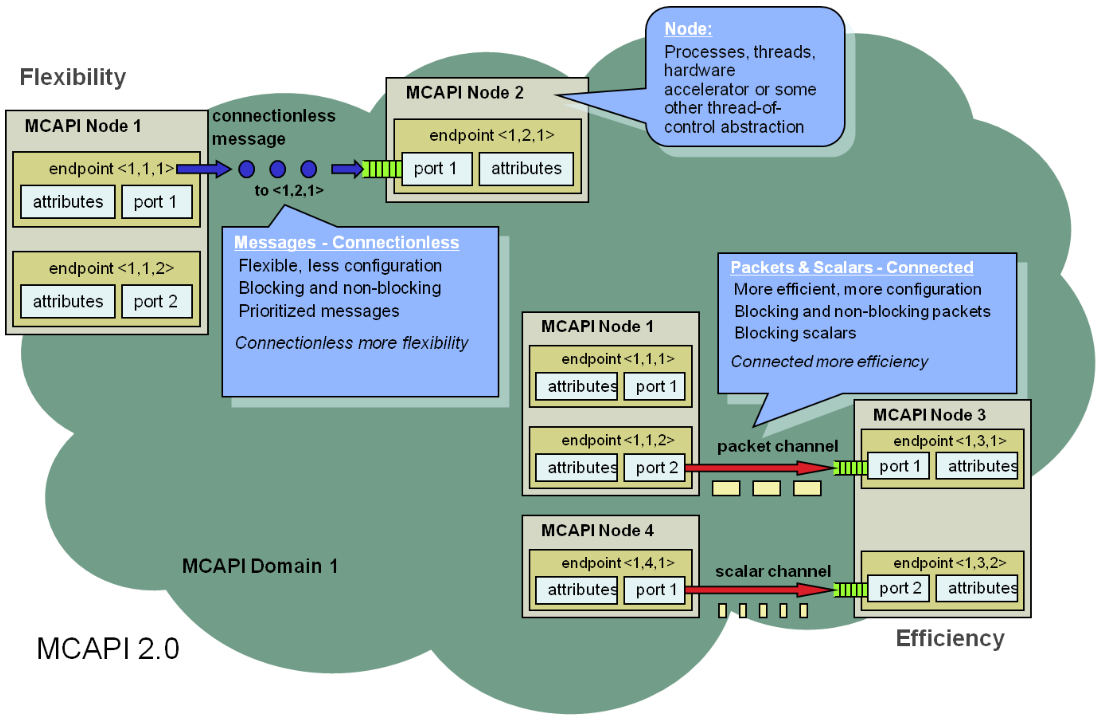
Some of the key concepts in MCAPI are:

Endpoints, the start and end points of the communication, defined by a topology unique 3-tuple <domain, node, port>.

Messages are connectionless, offers per message priority (FIFO order per priority level), buffered communication with blocking and non-blocking functionality.

Packet channels are connected, unidirectional with per channel priority, FIFO order,), buffered communication with blocking and non-blocking functionality.

Packet channels are connected, unidirectional with per channel priority, FIFO order,), scalar (8, 16, 32 or 64 bit) communication with blocking functionality.



3.4.2. MCAPI Enabling

To allow the function distributed across the cores to communicate, the functions are encapsulated with communication primitives.

Original application:

```
function_1();

function_2();

function_3();
```

Distributed application:

Core 1:

```
recv(); Receive data

function_1();

send(); Send result
```

Core 2:

```
recv(); Receive data

function_2();
```

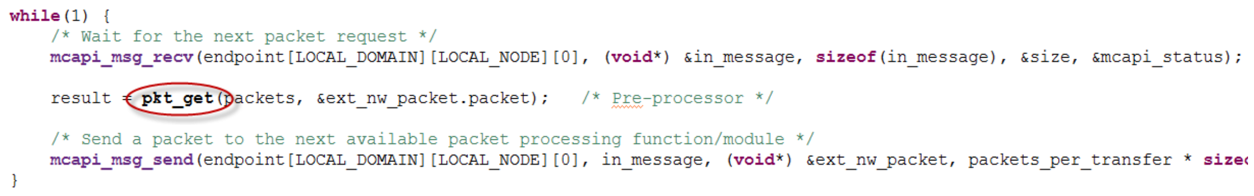
send(); Send result

Core 3:

recv(); Receive data

function_3();

send(); Send result



```
while(1) {
    /* Wait for the next packet request */
    mcapi_msg_recv(endpoint[LOCAL_DOMAIN][LOCAL_NODE][0], (void*) &in_message, sizeof(in_message), &size, &mcapi_status);

    result = pkt_get(packets, &ext_nw_packet.packet);   /* Pre-processor */

    /* Send a packet to the next available packet processing function/module */
    mcapi_msg_send(endpoint[LOCAL_DOMAIN][LOCAL_NODE][0], in_message, (void*) &ext_nw_packet, packets_per_transfer * size
}
```

3.5. OS Considerations

Multiple OS’es are being used as they provide different capabilities and allow this new platform to increase both its functionality and performance.

Linux provides access to configuration and management capabilities as well as rich-functionality networking stack. The RTOS provides high performance, predictable response times, configurability and scalability. It also allows us to use the same operating system for the ARM cores and the soft cores (and potentially also on the DSP's). The DSP OS has similar characteristics to the RTOS.

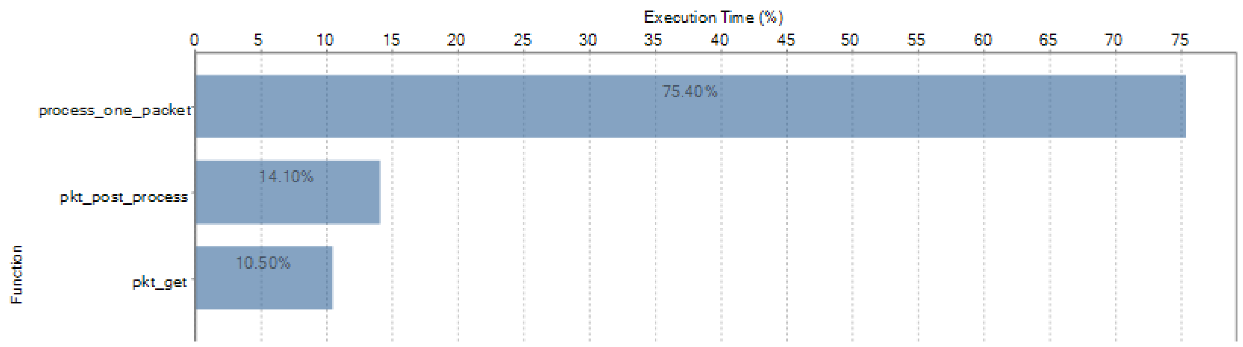
Of key importance is that the same communications programming model is used for the three OS'es.

4. Rapid Prototyping

As discussed above a number of factors come into play when migrating applications to multicore. It is very unlikely that the first assumptions based on the analysis and the multicore platform were perfect. A few or more iterations of analysis and refined assumptions will be required before an optimal performance is achieved and the functionality and performance requirements are met.

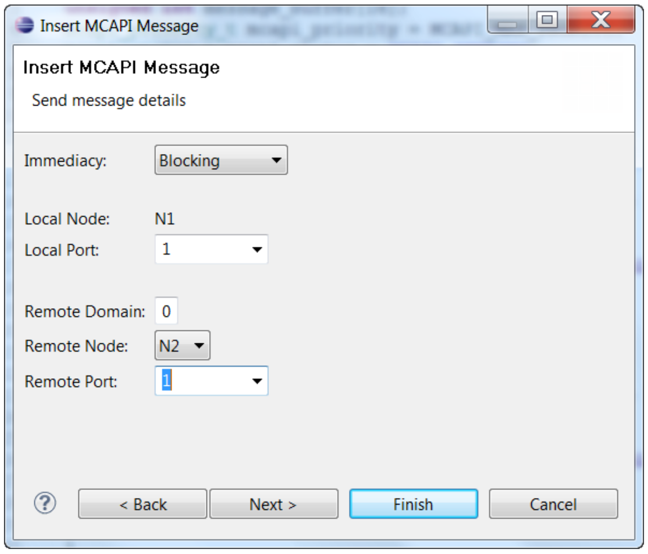
It is therefore important to have development tools that allows for rapid analysis, implementation, verification and optimization.

4.1. Analysis



4.2. Implementation

Application Level


Insert MCAPI Message

Insert MCAPI Message

Send message details

Immediacy: Blocking

Local Node: N1

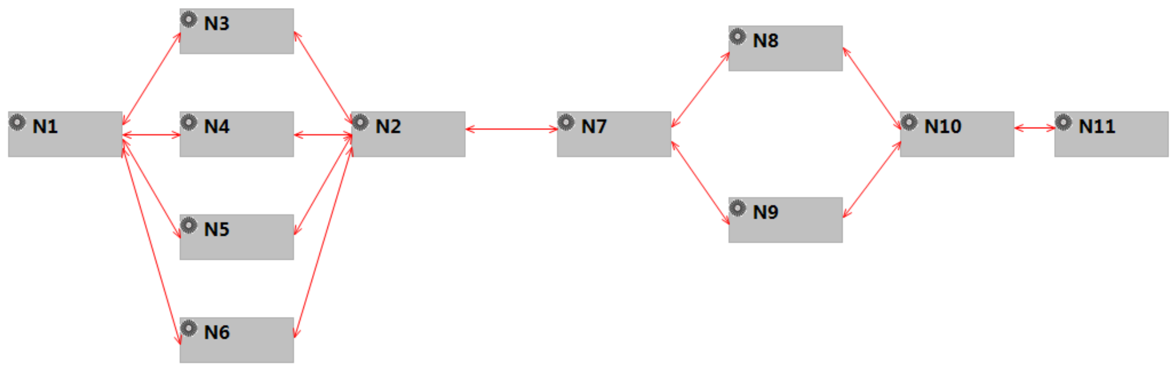
Local Port: 1

Remote Domain: 0

Remote Node: N2

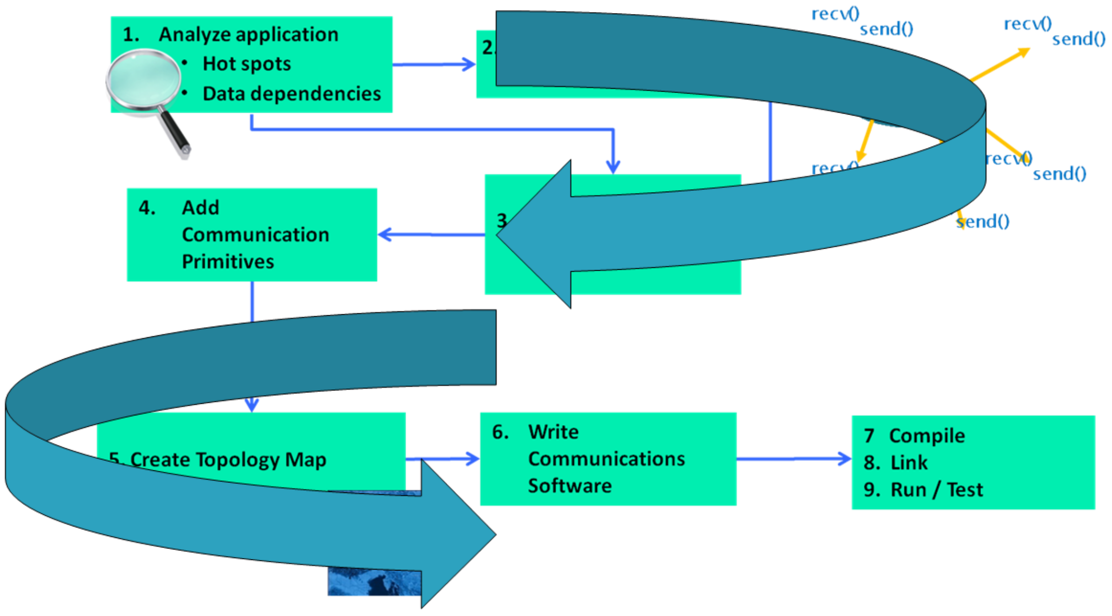
Remote Port: 1

Topology Level



4.3. Verification & Optimization

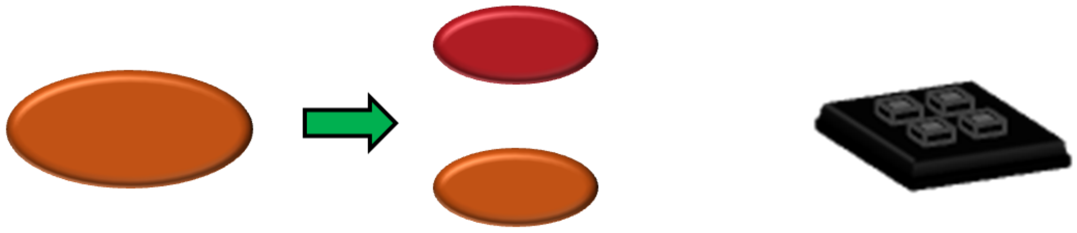
Reiterate until requirements are met.



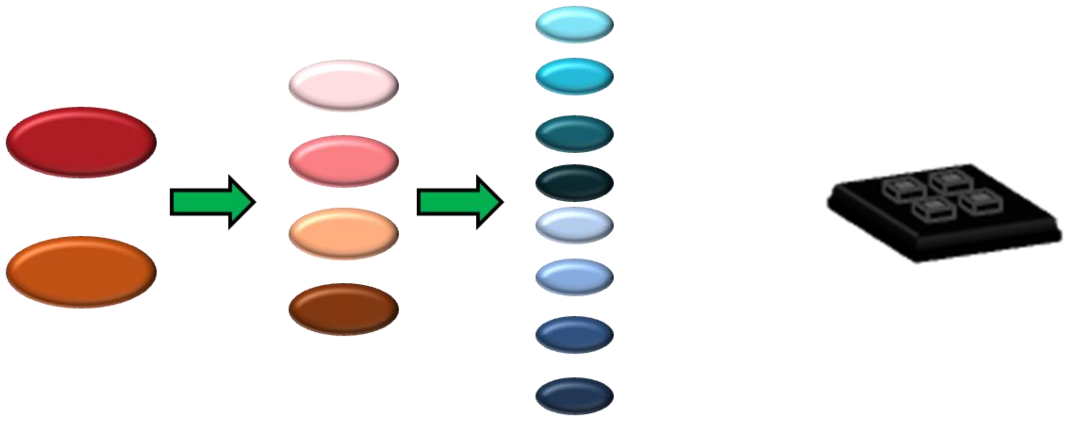
4.4. Migrate in Steps

It is often a good idea to migrate your application in steps. Start with the hotspots and divide in "big chunks" first. Once the first step is working, go to the next level and divide into smaller chunks. Having tools support makes it easier to migrate in steps.

Start with big chunks.



Refine.



5. Conclusions

We have discussed some steps that you can take to simplify migration your application to the ARM Cortex-A9 MPCore as well as how you can integrate with other parts in your multicore platform.

A unified standardized programming model for AMP and SMP will provide flexibility today and for next generation products, as it abstracts some of the complexities brought on by multicore.

Tools that allow rapid prototyping and exploration makes it easier to match the application to the multicore platform and they also makes it easier to migrate gradually.

Migrate the well known application blocks and improve performance through iteration.

6. References

[1] OpenMP - http://openmp.org/wp/
[2] MPI - http://en.wikipedia.org/wiki/Message_Passing_Interface
[3] Multicore - http://www.multicore-association.org
[4] MCAPI - http://www.multicore-association.org/workgroup/mcapi.php
[5] MRAPI - http://www.multicore-association.org/workgroup/mrapi.php
[6] AMP - http://en.wikipedia.org/wiki/Asymmetric_multiprocessing
[7] SMP - http://en.wikipedia.org/wiki/Symmetric_Multi-Processing