

# I. PROGRAMMING MODELS AND OS

---

## A. Programming models

Programmers have already started to face the problem of how maximize the performance of their applications running on multi-core/multi-thread processors.

Currently the limited power and thermal budget require programmers to change their programming models, from serial to parallel programming models. In fact, performance are mainly obtained through parallel computation: as the number of cores/threads in a single chip increases, the level of parallelism of the applications becomes a critical, limiting factor. In this view the interaction between the applications and the hardware is crucial to exploit the maximum level of parallelism of the application/hardware and, thus, to achieve the maximum performance with the lower power/energy consumption.

Several different parallel programming models have been developed and new ones are rising. In future programming models you foresee the need of more semantic passed by the programmer, transparent access to data, an easier programming environment for developing and debugging that provides automatic tools for parallelizing the application and interaction between the different levels of hardware/software stacks.

### A.1. Semantics

The programmer is the only agent, among the different components (compiler, run-time system, hardware, etc.) that interact during the development of an application, who knows about the real correctness and semantic of the application. This information are essential for the other components to properly perform their task extracting the maximum level of parallelism without reducing the correctness of the application. For example, without this information the compiler may not able to explore the maximum level of parallelism because it cannot determine whether parallelizing a loop may produce incorrect results. Another example is related to share data: it is risky to access this data in parallel without a synchronization mechanism because this may lead to data corruption.

In the future, we expect the programmers to provide more information about the applications, the data accessed and the possibility of exploring parallelism. This information can be provided at different levels:

1. The work assigned to the application can be divided in different parallel units (loops, filters, tasks, etc.), the programmer knows how to better divide the work among the different parallel units and, thus, can provide this information to the underneath components.
2. Shared data accessing: even if the application's code can be parallelized, bottlenecks can arise at run-time because of the contention of accessing data. Locking-based systems have been used until recently with good results, even if the burden of the synchronization mechanism was on the programmers. However, the increasing number

of cores/threads in modern and future processors leads to a bigger number of concurrent processes accessing the same data and the overhead and the burden of the synchronization increasing, making parallel application harder to write. New shared data access mechanism are under development and have shown promising results. Transactional Memory (with hardware, software and mixed approaches) is one of the most promising. With TM the programmer declares what are the critical section of the application without explicitly marking all the data structure that can, eventually, be accessed in parallel. The run time system (for software TM) or the hardware (for hardware TM) ensures the correctness of the execution flow and of the concurrent access to the shared data.

3. Input/output is currently a limiting factor for maximizing the level of parallelism in modern I/O-bound applications (web servers, DB servers, etc.). In fact, the I/O subsystem need to be accessed serially and do not provide support for roll-back in case the I/O transaction fails in some other point (for example, the process is not able to commit after issuing an I/O operation). Virtualization can provide a solution to this problem: with virtualization techniques, only virtual devices are exposed to the user and the I/O state of each device is kept until the process commits its transaction.

## **A.2. Transparent data access**

In future processors the cores/threads are likely to be grouped in clusters. This is mainly a consequence of the memory sub-systems, as memory does not scale so fast as processors. Thus, NUMA and distribute systems will be predominant. In those systems, the programmer has to properly allocate the application's data structure to minimize the memory latency. For example, the data used by a process should be allocated in the same NUMA-domain and the cost of migrating the data set should be considered whenever a process is about to be migrated from one node to another. All this complexities are today somehow handled by the run-time system, yet the programmer plays a big role in the allocation of data. In the future this complexity is going to increase with the number of NUMA-domains (being them chips inside a cluster node, a cluster node or a set of cluster nodes) and will limit the chance to explore more Memory Level Parallelism (MLP). Ideally, the programmers would like to access their data using a single name space even on distributed systems without manually handling data replacement, distribution, migration and, critical for exploiting parallelism, with the same synchronization mechanism regardless of data distribution.

## **A.3. Development Environment**

Writing and testing a parallel application is more complex than a serial application. As for serial code, programmers make use of development tools to speed up the time-to-market of their applications but, today, those tools are not nearly as good as the ones for serial code. Debugging, profiling, deadlock and race condition detection, memory leaks,

etc. are still activities mainly performed by hands, with a clear limitation to the development time of parallel applications. For example, stopping a parallel application and taking a snapshot of application's progress is not as easy as for a serial application (where tools like gdb are able to stop, restart, rollback or change on-the-fly the application), hence, the programmer has to include debugging information and code into the application source code. Moreover, an interactive environment is desirable to allow the user to perform activities such as restarting the application from a known state, changing the value of a variable at run time, etc.

#### **A.4. Automatic parallelization**

Ultimately, programming models and tools should provide programmers an easy way to write parallel applications, as if they were writing serial code. In order to do that, the tools (compiler, profiler, run time, etc.) should be able to automatically extract parallelism from the application's code, using the semantic previously expressed by the programmers through the programming model. The general idea is that the programmer expresses what can be parallelized and the programming tools select the best way to parallelize it.

Moreover, those tools should help programmers to understand what can be done in parallel and what cannot. Profilers, for example, can help the programmer to understand I/O flows and serialization point, providing information about the application's bottlenecks and how to increase the level of parallelism.

#### **A.5. Interoperability**

The problem of exploiting more parallelism from applications running on multi-core/multi-thread processors cannot be solved if attacked only at one level of the stack. Complex applications will probably make use of different programming models, compilers and programmers would like to run their applications on different kind of machines. This means that the hardware, the run time systems, the compiler, the programming tools and modes need to be able to talk with each other through well defined interfaces. For example, the compiler has to apply the directives provided by the programmer through the programming models primitives. In the same way, the compiler needs to interact with the run time system that will apply the correct decisions according to the run time environment.

## **B. Run time systems**

The run-time system is in charge of managing the hardware resources available in the system and of providing the user with an abstraction of those resources. As the complexity of the hardware resources increases (number of cores/threads per-chip, NUMA architectures, heterogeneous processors, etc.) the task of the run-time system increases as well. Future run-time systems should be able to efficiently manage the upcoming processors and to provide new services to the users (Quality of Service, fault tolerance, etc.).

### **B.1. Heterogeneity**

Several heterogeneous, multi-core processors have already appeared on the market, the IBM Cell being the most remarkable example. More heterogeneous multi-core processors with different kind of accelerators on-chip are expected in the future but how to manage all this different resources it is still an open problem. Specifically, one of the first problem to address is how to map computation on the cores/accelerators in order to maximize the system throughput. In fact, even if the users provide hints on the possible mapping computation-accelerators, some information will be only available at run-time. For example, having Tens or Hundreds of cores/accelerators in the system makes possible to run different applications at the same moment. At some point some of the processes belonging to two different applications may require the same resource, for example the same accelerators; in this case the run-time system should be able to handle those requests transparently to the user in the same way that current run-time system multiplex the CPUs or share the physical memory between the running processes.

Processors with multiple-clock domains also introduce a sort of heterogeneity in the CPU. In this case the run-time system has to manage this heterogeneity and properly map the workload on the different clock domains, for example, assigning the time critical computation to the faster clock domain.

### **B.2. Power- and temperature-aware run-time systems**

Power and temperature are currently some of the most limiting factors in the evolution of computing systems. In fact, having Hundreds of cores in the same chip/system does not necessary means that all of them can be used at the same moment. Power budget may impose that only a limited number of cores can be used at the same moment. Moreover, this number may depends on the current workload, as some functional units inside the processors consume more power than others. This kind of information is only available at run-time, when the run-time system is aware of the actual workload.

Even if the current power consumption is below the power budget, some part of the chip may become too hot if all the cores/accelerators in that zone are working at the same time. The run-time system has to manage this situation eventually migrating processes or computation from one accelerator to another (with similar capabilities) on another part of the chip (or another chip).

Other common solutions to reduce the power/temperature of a processors involves frequency scaling or clock gating some of the internal processor units. This solutions, as well as process migration, reduce the performance (for example, because of the reduced frequency speed or the cold-cache effect after a process migration) thus the run-time system has to balance performance and power/temperature, ideally maximizing the performance/watt ratio.

### **B.3. Virtualization**

Some applications do not scale after some point, which means that increasing the number of parallel processes (and assigning more cores to the application) does not produce a remarkable performance improvement. In this case it could be better to run more applications in the same system, for example, different customer applications in a data center.

In this scenario, virtualization plays an important role: in fact, customers do not want to share their environment with other users and prefer to run their applications inside a virtual machine. The run-time system provides the customer a virtual machine according to the customer's requests and constraints and ensures that this virtual environment is isolated from the others. Clearly, the underneath hardware resources are shared between the running virtual machines, though the users should not notice this, as the run-time system provides performance isolation (i.e., the performance of an application do not change regardless of the running workload). In order to guarantee performance isolation, some major changes are required at hardware, hypervisor and operating system level.

I/O is also critical in this scenario, as the I/O sub-system does not scale like the number of cores in the future processors. The virtualization environment should provide each user with an abstraction of the physical I/O device and multiplex the use of those resources among the virtual machines, transparently to the users and guaranteeing performance isolation.

### **B.4. Adaptive run-time systems**

The use of hardware accelerators instead of general purpose CPUs had proved to be efficient for both power and performance, especially if the accelerator is really specific and can perform its task quickly and with a limited power consumption. Those accelerators, essentially, perform in hardware common operations previously executed by software.

The main problem with this approach is that the amount of possible accelerators that can be included in a chip is limited and, therefore, the number of possible operations performed in hardware is limited too. Reconfigurable accelerators (like FPGA) are a possible solution to this problem. Those reconfigurable accelerators can be modified, with some limit in the reconfiguration, at run-time to provide different services to the users. Clearly, the reconfiguration does not come for free because the hardware require some time to adapt the accelerator for the new service.

It is desirable that the run-time system manages those accelerators threading off the cost of the reconfiguration with the possible benefit of executing some computation on the specific accelerator instead of on an general purpose processor. This thread off is in terms of both performance and power/temperature.

## **C. Quality of Service and Scalability**

### **C.1. Quality of Service**

It is clear how, in future systems, the number of parameters involved in the execution of an application will increase. Multi-core/thread processors share different resources at different levels while heterogeneous multi-core processors states that some resources are better than others. Finally, power- and temperature-aware system will limit the use of some of those resources. As a consequence the users may only be interested to execute their applications if the system can guarantee some level of Quality of Service (QoS). For example, a user may want to execute an application only if the system guarantees a minimum frequency speed; another customer may be more concerned about the power consumed; another one may require performance isolation, etc.

Regardless of the kind of resources, future systems should provide an interface to the users to express the level of Quality of Service and the resources they are interested to. Some example of resources and QoS levels are:

- core frequency (performance, power)
- accelerator/general purpose CPUs (performance)
- memory (performance, specifically critical for NUMA architectures)
- performance isolation

### **C.2. Scalability**

In the future, multi-core systems will not only scale up, going to big servers or supercomputers, but will also scale down to embedded systems like smart phones or media devices. In this view it is important that the same application runs efficiently on a large server as well as on a smart phones, using the same interfaces to talk with the underneath run-time system and hardware.

## **D. Possible evolution**

### **D.1. Next 5 years**

In the short term programmers have to face the problem of making applications parallel, most likely using the current programming models and tools. In the next 5 years we foresee that massive, large servers and supercomputers will still use the MPI programming model while small multi-core systems (single node, shared memory machine) will use POSIX threads or openMP.

Distributed applications and embedded systems (especially smart phones) are rapidly converging to well defined standards, for example, openweb for distributed systems and android for smart devices.

### **D.2. From 5 to 10 years**

Once the applications are running on multi-core systems, new programming models and solutions can be tested and implemented. We identified three major challenges in the medium term: single name space, extension of standards (like android) to multi-core systems, run-time systems capable of guaranteeing QoS. These challenges apply to embedded systems with small, low-power cores as well as to large clusters.