Models of Computation – Enabling Exascale

Thomas Sterling
Louisiana State University
May 17, 2009

The derivation of new systems' software and tools for high performance computing environments at Exascale will demand realignment and adjustment of functionality and capability of software components to exploit the new opportunities and address the new challenges of future system architectures, which themselves will be created in response to advancing hardware technologies. The evolution of digital device technology, the most dramatic in the history of human technology, has catalyzed a sequence of architecture classes over the last six decades, each optimized to the specific properties of their respective emergent technology phase. Programming models and representative languages followed to best exploit the performance capability of the system hardware during each phase. Algorithms were devised to reflect the computational needs of the applications while constrained to the semantic constructs of the available APIs. This reactionary strategy is being replayed as HPC once again experiences a phase-change with the advent of heterogeneous multicore for ultra-high performance computing. However, this empirical random-walk methodology is time consuming, error prone, and costly due to its intrinsic lack of guiding principles to facilitate co-design of all system layers simultaneously. Such comprehensive principles comprise a paradigm or *model of computation* to which all layers comply and contribute to achieve overall system optimal behavior with respect to critical objective functions. Can we get ahead of the game to leapfrog the tedium of catch-up? Or putting it another way, can a model of computation be derived that will enable the development of Exascale computer systems through the co-design of its comprising system software (and architecture) layers? A brief discussion of the nature and characteristics of models of computation (alternatively, "execution models") is offered to contribute to the current community discussions on proceeding toward the realization of Exascale computing by the end of the next decade.

Prior HPC phase-changes included the:

- original sequential instruction operation,
- sequential instruction issue,
- vector,
- array,
- systolic (for SPDs), and
- the most recent communicating sequential processes (CSP).

Others such as dataflow and reduction models did not achieve commercial status although interesting experiments were performed. The multiple-thread/shared-memory model is concurrent with CSP for limited scale systems.

The current HPC phase-change is apparent by the forced deployment of heterogeneous multicore components to maintain the continued peak performance progression consistent with Moore's Law and the underlying exponential growth in semiconductor device density. However, these structures are reactive to the combined pressures of power consumption, processor design complexity, and efficiency factors. They do not reflect a clear understanding of an underlying innovative execution model by which this combination of resources can be effectively employed for future applications. It is a subject of controversy as to whether incremental extensions to current methodologies (e.g., MPI) may serve this purpose. Four factors of the new phase suggest that incrementalism is a false hope even if it does adequately serve over the next three to five years with diminishing efficiency and scalability. These factors include:

1. > 1000X scalability gain with respect to current best levels
2. Power efficiency > 50 Gigaflops per watt,
3. Non-stop operation in the presence of single point failures, and
4. Support for efficient dynamic graph processing

Together these factors challenge conventional practices to:

a) Solve the multicore programming problem,
b) Reduce the ever increasing memory wall,
c) Expose and exploit billion-way parallelism,
d) Incorporate innate latency mitigation and hiding methods,
e) Reduce average energy per operation by two orders of magnitude,
f) Integrate memory-oriented operations for meta-data parallel computing,
g) Achieve fault tolerance through support at all levels,
h) Embrace dynamic adaptive resource management for runtime efficiency, load balancing, and reconfiguration (resiliency),
i) Exhibit a global address space,
j) Greatly increase efficiency of parallel control such as elimination of global barriers, lightweight task creation and context switching, and dynamic task migration, and
k) Permit heterogeneous cores to be optimally scheduled.

Other requirements may prevail as well but these are sufficient to demonstrate the inadequacy of common methods which over the prior decade and a half have resorted to static mapping of coarse grained parallelism to physical processes, avoiding latency rather than hiding it (noting some pre-fetch methods), overly constraining flow control by simplistic global barriers, forcing a distributed memory mind set, and forcing programmers to explicitly manage allocation of resources to data and tasks. No one layer of the system is sufficient to address any of these but multiple layers engaged synergistically implementing new strategies may do so. The model of computation provides the template for the patterns of execution to be accomplished each layer working in tandem with the others.

A model of computation describes how an abstract computation evolves on a physical machine successively altering the intermediate state of both to converge on a final solution. It defines the name spaces, the control semantics, the memory consistency model, the forms of parallelism that it may exploit, and potentially other attributes as well. It may define policy interfaces or invariants without specifying the actual specific policies themselves in order to provide flexibility in system implementation and application. Such policies might include scheduling methods and priorities, name space management, and means of achieving compound atomic operations for example.

There are multiple key consequences of adapting a model of computation to a new class of system hardware technologies. One is the verification through its existence and mapping of functionality requirements to hardware mechanisms that full and complex calculations can be performed on expected hardware designs. A second is that such a model simplifies overall system design. Without it, each layer of a system must be developed (assuming complete system design) in terms of every other layer; a order n-squared process. Adopting a model of computation only requires that each layer be defined in terms of its contributing functionality to realizing the shared model; basically an order n process. Even with iteration for convergent refinements and optimization, an execution model can greatly simplify the design process. A third value is that it does permit early experimentation with early algorithm and application kernels through the likely existence of a low-level application programming interface and test environments. While unlikely to provide absolute performance numbers, it will yield insight in to the utility of the control semantics of the model, and therefore future systems that employ it as a basis for hardware and software system design. And forth, such a model as has happened before, facilitates sharing and cooperation across disciplines and institutions.

How does a model of computation directly contribute to design concepts and decisions for the many combined layers of the system? Some examples, in no way comprehensive, are suggested:

- Application interface layer – the execution model defines the basic data organization, name space (shared or distributed), distributed communication semantics, and parallelism form and control. All these relate to the API and programming models that may be employed in constructing applications and libraries.

- Compiler layer – the model of computation combined with the system processors' ISAs and the previously defined API syntax determines the responsibilities in translation and analysis that is to be performed by the compiler. This includes invocation of runtime system functions and operating system service calls. The compiler will provide software implementation of software support mechanisms.
- Runtime system layer – A major effect of the model of computation is its definition of the functionality of the runtime system. This software is likely to grow in importance for new systems and will be heavily influenced by the model of computation determining how and what information about the runtime state will be exploited to manage tasks and resources. The runtime system will provide dynamic control, scheduling, allocation, and some synchronization of concurrent activities according to the underlying execution model.
- Operating system layer – the model of computation will determine what support it requires from the lower level system some of which will be provided by operating system services that must be provided.
- Architecture layer – For efficiency and scalability, the model of computation will require certain time critical mechanisms to be implemented at least in part in the hardware architecture to minimize overhead. Other architecture requirements driven by the model of computation include how to perform virtual to physical address translation, guaranteed compound atomic functions on data, and efficient communications.

Towards the establishment of the next generation model of computation, research is required to understand the driving requirements and to devise alternative solutions that will enable computing systems and methods for Exascale in the next decade. The above discussion has considered the general strategy and approach as well as the basic challenges that will guide the derivation of such a model of computation.