Modeling Execution Models (MEMS):

Progress Report for the X-Stack Meeting

Project Team

Pacific Northwest National Laboratory / PAL: Kevin Barker, Daniel Chavarria, Adolfy Hoisie (PI), Sriram Krishnamoorthy, Joseph Manzano, Abhinav Vishnu
Indiana University: Matthew Anderson, Thomas Sterling(PI)

Performance and Architecture Laboratory
Pacific Northwest National Laboratory

March 18, 2013

1 Introduction

As system architectures continue to advance towards Exascale, system and application developers require tools to enable reasoning about the impact on observable performance caused by design decisions. Often, this impact is difficult to predict at large scale given the multi-dimensional space under investigation and non-linear effects inherent in performance. Furthermore, researchers and designers need tools to quantify the trade-offs inherent in performance, power and energy consumption, and reliability. Application-centric analytical models enable a rapid exploration of both the architecture and system design spaces. Through an iterative process, we have successfully utilized models as a key component in the co-design process, enabling simultaneous exploration of both system and application design alternatives. For instance, models allow designers to quantify the impact of new hardware technology and configuration, as well as analyzing and guiding the application design, all in advance of system deployment. This capability allows designers to effectively explore the design space leading to more optimal solutions. Explicitly modeling the impact of the execution model on overall application behavior represents a new frontier in modeling. As architectures continue to increase in scale and complexity, execution models will evolve to enable efficient, high performance application execution. However, increasing system complexity brings with it new challenges for software and hardware architects and designers. The vastly higher degree of parallelism raises reliability concerns due to the greater number of system components while at the same time requiring applications to scale to levels of parallelism with lower threshold voltages never before encountered. Greater system size requires execution models that promote higher degrees of asynchrony as the cost of synchronization across all tasks in an application will be prohibitive. Above all, execution models must provide the means for reducing data movement, both within a “local” (e.g. inside a node) memory hierarchy and across the system. Underlying all of these concerns is the overarching optimization need in terms of energy consumption. As a result, using models to reason about application and system behavior, including through capturing the impact of execution models changes, is more important than ever. Models will enable designers to navigate the increasingly complex design space and provide insight into how applications utilize system resources. However, before we can expand analytical model frameworks with the notation of execution models, we need to be able to characterize such a concept.

This report is organized as follows. In section 2 we describe qualitatively the notion of an execution model. In section 3 we propose an analytical model frameworks with the notation of execution models, which allows us to characterize such a concept in a quantitative fashion. Section 4 concerns itself with the application workload under consideration and with modeling of these apps in terms of execution model characteristics. We summarize and draw conclusions in Section 5.

2 What is an Execution Model?

A simple definition is to consider an execution model as a layer that connects application and algorithms with the underlying hardware through its semantics. Other definitions includes the orchestration of computing on hardware and software resources and a paradigm of computing establishing the principles of computing that govern the inter-relationships of the abstract and physical components and their functions comprising the computational process. [2] Even though all these definitions seem to differ, they agree that there is an interaction between the hardware and software components of a system. Due to the different definitions, an easier way to describe (i.e. qualify) execution models would be to characterize them according to their features and characteristics. Efforts presented in [3] provide a methodology to characterize these models. However, current efforts cannot be connected to actual hardware implementations to measure (i.e. quantify) the impact of these execution model instances. Next, we present some examples of execution models to help illustrate this elusive concept[1].

2.1 Examples of Execution Models

Each generation of hardware advances provided new features that the software stack needed to take advantage of. Thus, it is not surprising that some of the biggest paradigm shifts in execution models coincide with evolutions in hardware technology. Here are some of the most well-known execution models throughout the history of computing:

Von Neumann Sequential Execution. The first of the execution models consists of the stored program principles, a single control path (single instruction pointer) and sequential fetch-execute-write execution cycles.

Vector Execution. Utilizing the concept of pipelining, vector execution models permit better utilization of the pipelined units and allowed the speed up of certain classes of applications. The advantages over the previous model include the overlapping of instructions and memory operations which resulted in better unit utilization and avoided resource hazards.

Dataflow execution model. This model re-organizes the computation on a set of primitive operations which are driven by the availability of data. This model exposes codes' maximum available parallelism and, following strict construction rules, can guarentee determinate concurrent execution.

SIMD Array Model. This model belongs to the class of Single Instruction Multiple Data models introduced by Michael J. Flynn in its famous taxonomy. Under this model, computation is performed by an array of simple identical components which execute the same instruction. The memory in this case is partitioned across all these components and operated upon by the array of processors.

Communicating Sequential Processes Model. This model is the most common used one today. Its semantics involve the idea of processes as concurrency units, private storage to each of these processes and "messages"[1] to communicate between them. Computation is carried out in each process and communication is carried out by messaging when needed. This model is an efficient fit for the Beowulf Clusters that grew to dominate the HPC landscape in the last decades.

Bulk Synchronous Parallel Model. Under this model, the underlying computation framework is still composed of a group of processes with private storage communicating via an interconnect network but it differs in the following. The computation is divided in a series of "super steps" in which the following three actions can take place: concurrent computation using the local memory; communication using single sided messages[2]; and an end-of-super-step barrier.

Global Memory Based Models. This umbrella term encompasses the execution models that have a global view of memory. This includes (but it is not limited to) flat shared memory models, Static Synchronous Global Memory Models, Dynamic Asynchronous Global Memory Models and ParalleX-like Models. Under the flat shared memory model, the memory is seen as a flat address range and it can be accessed by anyone at any time. Under the Static Synchronous global memory, each of the parallel entities can access the entire space but there is an affinity to a region of the memory which allows the exploitation of locality of references. In the case of Dynamic Asynchronous

[1]two sided messages are part of the model

[2]instead of the two sided ones from CSP

Global Memory models, the Static Synchronous Global Memory model is enhanced to allow the creation of both local and remote work[3].

Finally, the ParalleX execution model provides a global addressable memory space and fine grain synchronization constructs. The major components of this execution model include compute complexes, which are executing objects residing into a "Synchronous Domain"; ParalleX processes, which are containers for the complexes; Local Control Objects (LCOs), that provides polymorphic fine grain synchronization support; Parcels, which provide active message like semantics; and its Active Global Address Space, that allows a shared address space across the entire system's virtual addressing schemes. Other features like percolation and micro-check pointing are intrinsic to this model as well.

Both the qualification and quantification of these execution models is a critical aspect for the next generation and state-of-the-art systems of today. Algorithms are tailored to use execution model primitives in an efficient manner and execution model runtime must adapt and efficiently use the hardware resources available to them. Thanks to co-design, the idea of tailoring application, execution models and hardware features to maximize a specific metric (power, performance, resilience or a combination of thereof) is taking hold in the High Performance Computing Community. However, without a clear view of what each of these components contribute to the entire picture; we are just searching blindly in a very large space without a clear direction. That is why, characterizing, both qualitative and quantitative, execution models allows the community to understand the current state of affairs and the need (or lack thereof) for new execution models. Moreover, this allows the creation of predictive models for new execution models and new architectures. Such methodology for characterizing execution models is represented in the SCaLeM / AntiCiPate framework which is explained next.

3 SCaLeM and AntiCiPate: An Introduction

The first step in modeling the behavior of execution models is a characterization of the key properties that define the execution model. We consider the following four properties in identifying an execution model: nature of Concurrency, Synchronization between concurrent units, the nature of Memory, including memory consistency, and expression of Locality. These four

[3]The most well-known implementations of these models can be found in the runtimes of PGAS and APGAS programming languages, like Global Arrays for PGAS and X10 for APAGS

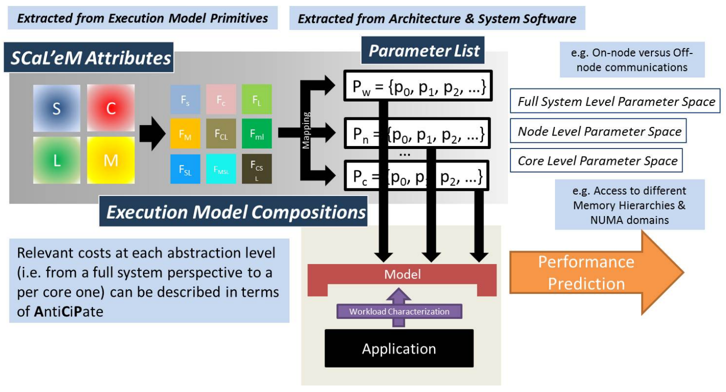

Extracted from Execution Model Primitives
Extracted from Architecture & System Software
SCaL'eM Attributes
S
C
L
M
F_S
F_C
F_L
F_M
F_CL
F_ml
F_SL
F_MSL
F_CSL
Mapping
Parameter List
P_w = {p_0, p_1, p_2, …}
P_n = {p_0, p_1, p_2, …}
…
P_c = {p_0, p_1, p_2, …}
e.g. On-node versus Off-node communications
Full System Level Parameter Space
Node Level Parameter Space
Core Level Parameter Space
e.g. Access to different Memory Hierarchies & NUMA domains
Execution Model Compositions
Relevant costs at each abstraction level (i.e. from a full system perspective to a per core one) can be described in terms of AntiCiPate
Model
Workload Characterization
Application
Performance Prediction

Figure 1: The SCaLeM and AntiCiPate Framework

attributes synchronization, concurrency, locality, and memory are referred to as SCaLeM for short. In addition to being fundamental, these attributes also identify properties of execution models have a quantifiable impact on the runtime behavior of programs written using a given execution model. In doing do, these attributes identify the key properties of an execution model, taking us from an abstract characterization towards concrete definition and characterization.

While these attributes identify useful characteristics, they are not sufficient in fully understanding the runtime behavior of an execution model. For example, the interaction between concurrency units and data locality properties has crucial implications on efficient program execution. We therefore consider *compositions* of these attributes that help in denoting properties that cannot be individually associated with any one attribute. The above relation is associated with the composition between concurrency and locality.

Note that attributes and compositions primarily catalog the basic properties of a given execution model. In particular, they identify the features of an execution model a programmer would be most interested in when developing optimized application using the given model. We separate the information associated with concrete instantiation of an execution model on a given architecture. These, referred to as *parameters*, identify primitives corresponding to the expression of the attributes and composition and associate models for each target implementation. The methodology involved in deriving models within the space defined by Attributes, Compositions and Parameters forms the AntiCiPate framework. Although not described in this short report, SCaLeM and AntiCiPate allow for an unambigiguous definition of what an execution model is and a quantitative distinction from other execution models cast within this framework. Once the parameters have been quantified, runtime behavior of applications written using an execution model can be validated on current systems. The validated models can then be analyzed in terms of their sensitivity to various parameters on future systems and future incarnations of the execution model of choice. A graphical representation of this process is given by Figure 1.

The practical usefulness of specific execution models in solving exascale problems depends on the degree of difficulty (or ease) with which extreme-scale applications can be written using them, and the extent of our understanding in realizing these execution models on the target architectures. Alternatively, an exascale system architecture should efficiently support the most promising execution models. Our framework enables comparative analysis of and reasoning about execution models in answering these questions. Attributes and compositions help identify the similarities and differences between execution models at a quick glance. An execution model can be easily supported on a target system if the approaches to effectively implementing its key components are well understood. A group of closely related execution models can be supported on a given system architecture with much greater ease, and possibly lower cost, than diverse models.

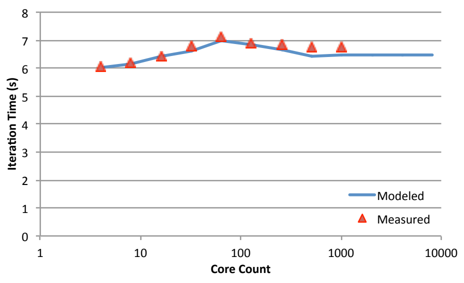
We are in the process of cataloging the key properties of execution models such as the ones listed above, and mapping them to the SCaLeM attributes and their compositions. The primitives in representative programming models for each execution model (e.g., MPI for CSP) are mapped to the parameters, which are then modeled using the AntiCiPate framework. As application modules are available on each execution model of interest, we are working towards modeling their behavior and analyzing their sensitivity to changes in the parameters.

4 Application Descriptions and Modeling Results

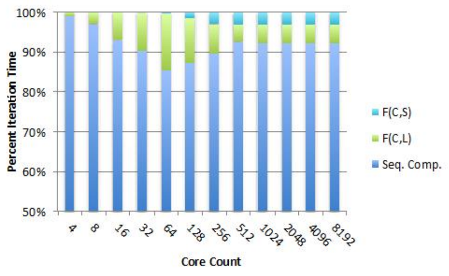
In this section, we introduce one example of an application that has been studied: GTC. Further, we describe modeled results that have been obtained and a brief sensitivity analysis in which application performance impact due to changes in the costs associated with SCaLeM compositions is quantified.

Gyrokinetic Toroidal Code (GTC)

GTC is a 3-dimensional code used to study microturbulence in magnetically constrained toroidal fusion plasmas. GTC is a particle-in-cell code which solves equations describing the evolution of a system of particles under the effects of self-consistent electromagnetic fields. The unknown in these equations is the flux, which is a function of time, location, and particle velocity and represents the distribution function of particles in phase space. The normal mode of operation for GTC is weak-scaling, where the number of particles per process is configurable and is an input parameter into the application model. Figure 2 depicts both measured vs. modeled application



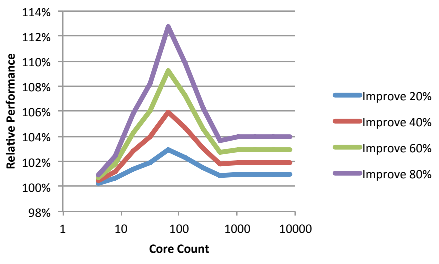
(a) Model validation



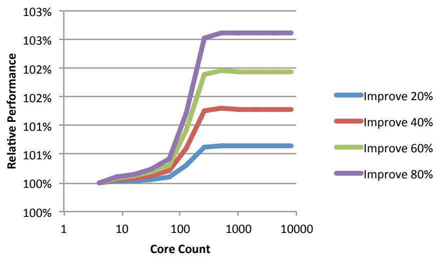
(b) Breakdown of modeled runtime

Figure 2: GTC Model Results

performance on an AMD Opteron cluster connected via InfiniBand and a breakdown of modeled performance in terms of the compositions defined by the SCaLeM framework. From Figure 2(a), it can be seen that the GTC application model developed is quite accurate, showing an error of less than 5% to 1,024 cores. Interestingly, the execution time can be seen to reach a peak at 64 cores before flattening again (runtime actually increases slightly at larger scales due to collective MPI operations called during execution). This is caused by TLB miss penalties; up to 64 cores, the toroidal tokamak domain is partitioned using a one-dimensional radial method. Critical data structures are accessed using a strided pattern in which the stride is dependent on the number of radial partitions. As this stride grows, page faults and resulting TLB miss penalties have a major impact on performance. Beyond 64 cores, the partitioning method expands along an orthogonal dimension, limiting the impact on performance. Figure 2(b) shows the breakdown of the modeled GTC performance and indicates the relative cost of local memory access (including TLB miss penalty) increases to 64 cores (shown by $F(C, L)$, since it has aspects of both locality and memory attributes). At larger scales, local memory access represents less of an impact on overall application performance; however the cost of interprocessor synchronization, represented by all MPI communication operations, increases (shown by $F(C, S)$, since it has aspects of both concurrency and synchronization attributes). This cost will continue to increase with scale due to the presence of global synchronization operations.



(a) GTC sensitivity to $F(C, L)$



(b) GTC sensitivity to $F(C, S)$

Figure 3: Modeled Sensitivity Analysis

Sensitivity Analysis

Given a validated and trusted performance model, it is then possible to quantify the impact on overall application performance caused by variations in the cost of each SCaLeM composition. Ultimately, this will be extended to provide a mapping between two distinct execution models. However, several caveats exist which preclude undertaking such an activity blindly: in particular, it must be noted that the execution model imposes constraints on any given algorithm and its implementation. By moving from one execution model to another, the algorithms which comprise the application will not remain constant and, in some cases, the application must be fundamentally reworked.

Figure 3 depicts the modeled performance impact of varying the cost of the $F(C, L)$ and $F(C, S)$ compositions on the GTC application. This application shows a greater sensitivity to the $F(C, L)$ composition, resulting from a sensitivity to intra-node memory contention.

5 Conclusions and Future Work

The next steps in the modeling work require modeling applications that are not implemented in the CSP execution model. Ideally, the modeling methodology we have developed and implemented will be applicable to any application implemented on any execution model. However, it is necessary to ensure that the techniques employed capture all of the salient features of an application executing on any software platform, and that features provided by the hardware/software stack map to attributes and compositions defined by SCaLeM. Furthermore, although we have described results that apply only to performance, the methodology described and employed should be applicable to any metric of interest (e.g., power/energy and reliability).

Once experience has been gained in modeling non-CSP execution models, the goal is to use modeling methods to map applications between execution models. Models will be the tool that allows for quantitative comparison between execution models on hardware platforms of varying capability. As mentioned, this necessarily implies that applications and algorithms will undergo fundamental changes, which may not be predicted by a model. However, through experience and reasoning about how an application will utilize hardware and execution model features, an application-specific model will be able to capture these changes.

To facilitate the characterization of more execution models, rules for the construction of each of the compositions and the attributes will be formalized. Moreover, the mapping operations and parameter's cost functions will be expanded to include more architecture configurations.

This report has shown the SCaLeM and AntiCiPate framework used to characterize execution models qualitatively and quantitatively. Concepts like attributes, compositions, and architectural / system parameters were introduced. Finally an example application and execution model were selected and analyzed on this framework.. Using these types of modeling frameworks, next generation systems can be designed with all aspects of the computation spectrum in mind such that the maximum performance can be extracted.

References

[1] Kevin Barker, Daniel Chavarria, Adolfy Hoisie, Darren Kerbyson, Joseph Manzano, Vishnu. Abhinav, Matthew Anderson, and Thomas Sterling. Modeling execution models, 2012.

[2] Hartmut Kaiser, Maciek Brodowicz, and Thomas Sterling. Parallex an advanced parallel execution model for scaling-impaired applications. In *Proceedings of the 2009 International Conference on Parallel Processing Workshops*, ICPPW '09, pages 394–401, Washington, DC, USA, 2009. IEEE Computer Society.

[3] Thomas Sterling. Doe abstract models for exascale computing, 2011.