Towards Exascale Co-design in a Runtime System

Thomas Sterling, Matthew Anderson, P. Kevin Bohan, Maciej Brodowicz, Abhishek Kulkarni, Bo Zhang

Center for Research in Extreme Scale Technologies,
School of Informatics and Computing,
Indiana University, Bloomington Indiana

Abstract. Achieving the performance potential of an Exascale machine depends on realizing both operational efficiency and scalability in high performance computing applications. This requirement has motivated the emergence of several new programming models which emphasize fine and medium grain task parallelism in order to address the aggravating effects of asynchrony at scale. The performance modeling of Exascale systems for these programming models requires the development of fundamentally new approaches due to the demands of both scale and complexity. This work presents a performance modeling case study of the Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics (LULESH) proxy application where the performance modeling approach has been incorporated directly into a runtime system with two modalities of operation: computation and performance modeling simulation. The runtime system exposes performance sensitivies and projects operation to larger scales while also realizing the benefits of removing global barriers and extracting more parallelism from LULESH. Comparisons between the computation and performance modeling simulation results are presented.

1 Introduction

Understanding and managing asynchrony effects in simulating Exascale parallel machines with eventual billion-way parallelism is a crucial factor in achieving application efficiency and scalability. Efforts to manage asynchrony have resulted in the creation of a number of emerging programming models and the renovation of several traditional programming models all with the aim to utilize asynchrony and extract more parallelism from applications at large scale. A key component of these efforts is performance modeling.

Several performance models have been created specifically to highlight shortcomings in how traditional programming models fail to adequately address asynchrony. One of these is the Starvation-Latency-Overhead-Waiting for Contention (SLOW) performance model [14] where each letter of the acronym SLOW identifies one of the key causes for constrained scalability in an application and highlights a challenge when programming using the conventional practice. Several alternatives to conventional practice have been developed to better address the issues highlighted by SLOW by utilitizing lightweight concurrent threads

managed using synchronization primitives such as dataflow and futures in order to alter the application flow structure from being message-passing to becoming message-driven.

However, the performance modeling necessary to understand and manage asynchrony effects at scale can be especially challenging for emerging programming models that rely on lightweight concurrent threads. Trace-driven approaches for such programming models tend to substantially alter the application execution path itself while cycle-accurate simulations tend to be too expensive for co-design efforts. While discrete event simulators have been successfully used for the performance modeling of many-tasking execution models before [4], they require both an implementation of the execution model in the simulator as well as a skeleton application implementation. This skeleton code has to preserve the dataflow of the original application while appropriately modeling the computational costs of the full application in between communication requests.

A robust implementation of the execution model in the discrete event simulator and a close representation between the skeleton code and the full application are both crucial in order to achieve accurate performance predictions from the discrete event simulator. A skeleton code which closely represents the computational costs and dataflow of the full application code can be especially difficult to achieve because a significant code fork is necessary in order to develop the skeleton code. Updates and improvements made to the full application code are not automatically reflected in the skeleton code and inconsistencies between the two codes are easily introduced. Likewise, accuracy in implementing the execution model in the event simulator is also difficult to achieve: modeling the contention on resources, the variable overheads when using concurrent threads, the highly variable communication incidence rates, the network latency hiding, the thread schedulers and associated contention, and the oversubscription behavior all contribute in complicating the implementation of the execution model in the discrete event simulator.

This paper presents a performance modeling case study for many-tasking execution models which incorporates performance modeling directly into the runtime system implementation of the execution model without requiring a skeleton code or application traces. A runtime system is the best equipped tool for performance modeling an application as it comes with the necessary introspection capability, it does not require a skeleton code separate from the application for modeling, and is itself already a robust implementation of the execution model it represents. For this case study, the performance modeling capability of the HPX-5 runtime system is explored for a proxy application developed by one of the US Department of Energy co-design centers: the Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics (LULESH) proxy application code [1]. LULESH has been ported to multiple programming models, both emerging and traditional, and its scaling behavior has been extensively explored making it a good candidate for this case study. More importantly, the scientific kernel encapsulated in the LULESH proxy application is expected to be representative of computational science applications requiring future Exascale resources.

The HPX-5 runtime system is an implementation of the ParalleX execution model [9] and supports message-driven computation as well as two different modalities of operation: full computation and performance modeling simulation, hereafter referred to as simulation. The simulation modality in this case study is restricted to those cases where the prototype Exascale node is already available for simulation. This enables the runtime system to produce performance predictions for large systems composed of those prototype nodes. This approach that does not require a separate skeleton code nor code tracing instrumentation for use in performance studies and application co-design.

Overall, this work provides the following new contributions:

- It presents a port of LULESH proxy application to the ParalleX execution model.
- It presents a performance modeling approach for many-tasking execution models where the performance modeling has been incorporated into the runtime system.
- It presents a performance modeling approach that is not trace-driven and does not require a skeleton code.
- It explores a runtime system with two modalities of operation for both performance modeling simulation and full computation operation.

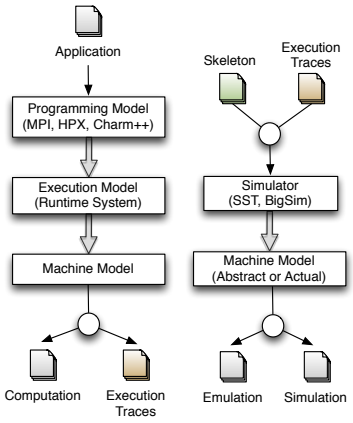
This work is structured as follows. Related work is given in Section 2, followed by a description of the performance modeling approach proposed here. Details about the runtime system used in the case study are given in Section 4 along with motivation why modern runtime systems are well suited for performance modeling when using a many-tasking execution model. Secton 5 gives details about the HPX-5 implementation of LULESH used here while Section 6 presents the results of the LULESH case study. Our conclusions and directions for future work are given in Section 7.

2 Related Work

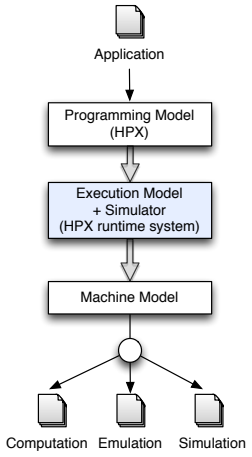
While there have been a large number of approaches to developing performance modeling techniques which are application independent, most of these have centered around the Communicating Sequential Processes (CSP) execution model. Trace-driven approaches are a key component in many performance modeling and co-design frameworks, including DUMPI in SST/Macro [11], LogGOPSim [10], and the Performance Modeling and Characterization (PMaC) framework [6]. A key challenge in trace-driven approaches is the trace collection overhead. Carrington et al. [5] demonstrate how to reduce the trace collection overhead by extrapolating results to larger core count sizes using smaller core count traces. While trace-based approaches generally do not require changes to the user application and work well with the coarse-grained computation style favored by CSP, trace collection overhead can significantly alter the execution path for the fine-grained computation style favored in many-tasking execution models.

A domain specific language (DSL) approach to performance modeling was introduced by Spafford et al. [13], named Aspen. Aspen provides a common set of tools and concepts to more easily enable coarse-grained exploration of algorithms and co-design. However, Aspen also makes some limiting assumptions which could prevent fine-grained, message-driven style computations. To work around such shortcomings, the message-driven toolkit Charm++provides its own trace-driven parallel discrete event simulator, BigSim [15], which is itself capable of parallel computation. When used in conjunction with the Charm++performance emulator, BigSim Emulator [16], coarse timing predictions can be made to guide co-design decisions. As it is a trace-driven approach, the traces can impact the execution path of sufficiently fine-grained computations.

In the context of fine-grained computations with significant resource over-subscription, performance modeling options for many-tasking execution models are very few and, up to now, require skeleton code creation in order to avoid trace-driven approaches. Sottile et al. [12] present a semi-automatic way of extracting software skeletons using source-to-source code generation as one way to avoid forking application codes for discrete event simulation. Robust, generic, and fully automatic approaches for skeleton code generation are difficult to find.



(a) Traditional simulation approaches are either trace-driven or skeleton code driven making the application one step removed from the simulator.



(b) An alternative approach takes advantage of the increasing runtime system introspection available to add simulation capability directly to the runtime.

Fig. 1: An illustration comparing the traditional performance modeling approaches with what is proposed here.

3 Performance Modeling

This case study targets performance modeling scenarios where the actual or prototype Exascale node is available. Unlike traditional simulation approaches, the proposed simulation methodology does not involve generating traces nor a skeleton code but rather integrating the simulation capability with the runtime system. Figure 1 highlights the differences between traditional simulation approaches and what is proposed here. This alternative approach is motivated by the goal of improving user access to performance modeling, the rapid increase in the number of many-tasking execution models, and the ability for modern runtime systems to incorporate all necessary introspection mechanisms to properly operate in a performance modeling simulation mode. Further motivation as to why the runtime system is well suited for this type of modeling is provided in Section 4.

For any application, the many-tasking runtime system has full and direct access to the task phase information. When the runtime is operating in a simulation modality on prototype Exascale nodes, a sample of nodes is selected for performing the application simulation. Other nodes that directly interact with these nodes are also simulated but only for a small set of communication iterations to provide accurate message incidence rates for the sample nodes. Using select iteration snapshots in the course of the application simulation, the runtime system uses these sample nodes to predict application performance at the scale indicated by the user. While this approach does not require traces, it has a disadvantage of not providing performance predictions for the entire duration of application execution. The performance predictions are provided only for a specific subset of communication iterations.

The approach is illustrated in Figure 2. Each square and circle represents a node in an Exascale simulation while arrows indicate communication. When the runtime system is in simulation mode, a user defined set of sample nodes, indicated by the red outlined boxes, is selected for running the application. Nodes which interact with this sample set, indicated by blue circles, are identified by the runtime system accessing the node interaction data. The application is also run on these nodes in order to provide correct incidence rates and phase information to the sample nodes but their runtime information, such as specific execution times of various subroutines, is not used in the performance prediction. Network communication is performed between all nodes that are running the application while a network model handles communication between circle nodes and non-running ghost nodes, indicated by blue squares. Green arrows indicate network traffic approximated by a network model, red arrows indicate real network traffic, and black arrows indicate traffic not modeled.

The accuracy of the predictions relies on how well the sample nodes represent the overall state of the application. For static dataflow applications which are well-balanced, this would be easily achieved with a very small sample size. For highly dynamic applications, it would not be unlikely to require terascale computing in order to predict Exascale performance.

This runtime system based approach can be improved and refined in several ways. The number of buffer nodes which provide incidence rate and node interaction information to the sample set can be increased to improve accuracy. Likewise, the introspection capability of the runtime system can be expanded to directly model these phases and incidence rates while in full computation mode and then later re-used in the simulation modality while still avoiding trace collection. In this case study, we present results from the simplest performance modeling approach where sample nodes operate in full computation mode with all other nodes operating as ghost (non-computing) nodes. The following section gives details about the runtime system selected for this case study and how runtime system capabilites are well suited for taking on the role of performance modeling.

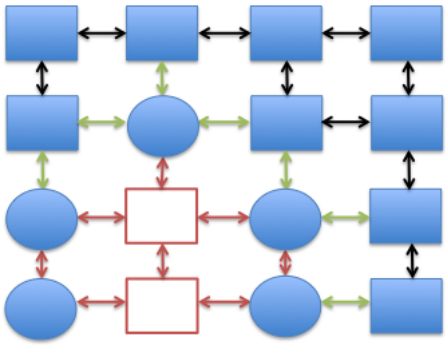


Fig. 2: A runtime system based performance modeling approach. Each square and circle represents a node in an Exascale simulation while arrows indicate communication. When the runtime system is in simulation mode, a user defined set of sample nodes, indicated by the red outlined boxes, is selected for running the application. Nodes which interact with this sample set, indicated by blue circles, are identified by the runtime system accessing the node interaction data. The application is also run on these nodes in order to provide correct incidence rates and phase information to the sample nodes but their runtime information is not used in the performance prediction. Network communication is performed between all nodes that are running the application while a network model handles communication between circle nodes and non-running ghost nodes, indicated by blue squares. Green arrows indicate network traffic modeled by a network model, red arrows indicate real network traffic, and black arrows indicate traffic not modeled.

4 Runtime Systems and Performance Modeling

Performance modeling of Exascale systems requires development of fundamentally new approaches due to demands of both scale and complexity. The trace based methodologies are infamous for generating prohibitively large volumes of data when run on many nodes of a large system, necessitating the use of the on-the-fly compression that potentially distorts timing, or decimation of data, which reduces overall accuracy. Full scale fine-grain discrete event simulation may easily exceed the application's run time on the actual hardware. The skeleton based approximations may result in faster simulation, but also tend to reduce the accuracy due to overly simplified models of execution resources, memory, and network, as well as their interactions. To address these shortcomings, an approach inspired by and integrated with the model of execution is proposed. Unlike most existing solutions that necessarily restrict their functionality to a single or at most a few layers of system software stack, execution model spans the whole gamut of software services and underlying hardware, permitting more thorough analysis. The ParalleX execution model and its associated HPX-5 runtime system implementation are used for this case study.

ParalleX is a new model of execution explicitly created to identify and mitigate the effects of primary sources of performance degradation in parallel applications. They include: (i) **S**tarvation, or insufficient amount of work necessary to efficiently utilize the available execution resources, (ii) **L**atency, or delay in accessing remote resources and services, (iii) **O**verhead, or additional work required for management of parallel computations and resource allocation on critical path, but absent from the sequential variant, and (iv) **W**aiting for resolution of contention on concurrently accessed resources and services. The newly added extensions of the ParalleX model deal with **E**nergy efficiency of computation and its **R**esilience, or achieving reliable execution in the presence of faults (SLOWER). ParalleX addresses many limitations of commonly used application programming models such as MPI, by breaking free of Communicating Sequential Processes scheme (which frequently results in overly constrained implementations abusing global barriers). Instead ParalleX relies on message-driven approach that avoids predetermined patterns of interaction by combining lightweight threads, fine-grain synchronization, and active messages called *parcels*.

Even though some of the model components have been known for more than a decade, ParalleX organizes them into a novel parallel execution framework with unified semantics. The system is subdivided into a number of localities, or physical resources with bounded service response time. In typical platforms (clusters, constellations), locality corresponds to a computational node. The localities are connected by asynchronously operating network. Application state may be arbitrarily distributed across any number of localities in the system. Local modifications of application state are carried out by threads. In ParalleX, threads are by definition ephemeral, created for and existing only long enough to execute a specific task. This makes them a convenient medium to represent the unconstrained parallelism available in the application. Thread execution is syn-

chronized by Local Control Objects (LCOs). These structures implement high-level synchronization primitives, such as futures or dataflow elements, although support of traditional atomic operations is also possible. Both threads and LCOs are closely integrated with scheduling algorithms to permit event-driven operation (and avoiding busy-waits and polling) as much as possible. Threads and LCOs along with related data structures can be embedded in ParalleX processes — entities that hierarchically organize parallel computation and provide logical encapsulation for its individual components. Unlike UNIX processes, they can span multiple localities (and therefore multiple address spaces). Processes, threads, and LCOs may migrate between the nodes and are globally addressable, permitting the programmer to access them from anywhere in the system. This is controlled by the Active Global Address Space (AGAS), a distributed service that maintains lookup tables storing physical locations of all first class objects of the computation. ParalleX functionality manifests itself primarily in the runtime system layer, which, through its proximity to the application code permits additional optimizations and acts as an intermediate layer for access to expensive (in terms of overhead and latency) OS kernel services. ParalleX compliant runtime system implements introspection, supporting direct access to integrated performance counters and enabling monitoring of application activity. This is particularly valuable for low overhead collection of performance data.

HPX-5 is a high performance runtime system that implements the ParalleX model, providing the ability to run HPC applications at-scale and to simulate the performance characteristics of code without actually fully running the application.

Written in C and assembly, the HPX-5 runtime system is focused primarily on algorithmic correctness, performance, and stability. To achieve this, HPX-5 is developed with an extensive suite of tests that execute well known scientific codes with published results and uses these to ensure correctness and stability.

The runtime is highly modular and is comprised of several components, including:

- A user-space *thread manager* made up of M:N coroutines similar to Python Green Threads. HPX-5 threads are continously rebalanced across logical CPU cores in a NUMA-aware way that ensures a high degree of continuous work.
- An asynchronous network layer built on RDMA *verbs* capable of running on InfiniBand, Cray Gemini, and Ethernet networks as well as in a non-networked (SMP) environment.
- A *parcel dispatch* system that routes messages between objects and makes runtime optimizations through direct integration with the node's network interface controller (NIC).
- A variety of distributed lock-free control structures, including *futures* and logical gates that provide programmers with an easy-to-use environment in which to define application dataflow.
- An *active global address space* (AGAS) that automatically distributes and balances data across all nodes in an HPC system.

– Support for multi-core embedded architectures (such as ARM).
– Instrumentation to perform simulations of application runs in a variety of environments, using *spec files* that describe several well-known machines.

In addition to normal operation, the HPX-5 runtime supports a *simulation* mode in which it models performance of a full (non-skeleton) computation application as it would run on a target system. It achieves this in two ways: a) by directly modeling performance on the target system (simulation), and b) by emulating the performance of system other than the one it is running on through the use of pre-generated specification files that detail the typical performance characteristics of the target system's hardware as well as interconnect network topology and other features.

Using the ParalleX execution model, the MPI based LULESH proxy application has been ported to the HPX-5 runtime system. The following section briefly describes this port and how it differs from the MPI implementation.

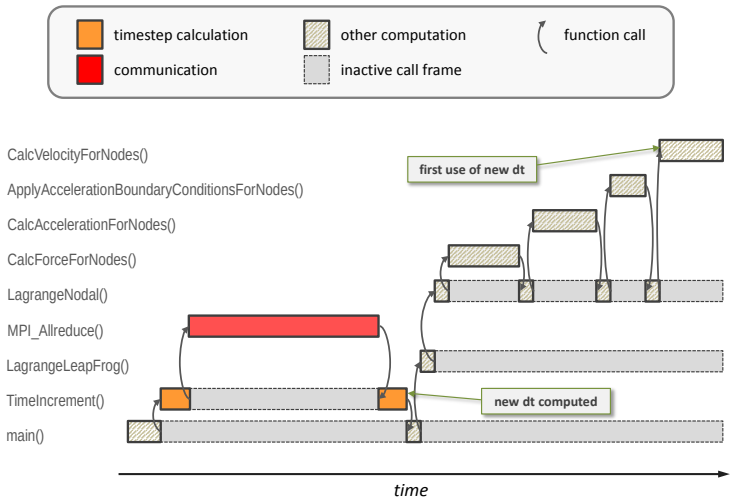
5 ParalleX LULESH

The implementation of LULESH in ParalleX is optimized by removing global barrier calls like `Allreduce` and overlapping the communcation needed for the reduction operation with computation. Owing to this, ParalleX is able to extract some performance benefits over the original MPI implementation of LULESH. The HPX-5 implementation of the LULESH application is based on the same domain-element hierarchy employed in the MPI version available at [2]. But it differs from the MPI implementation in three aspects.

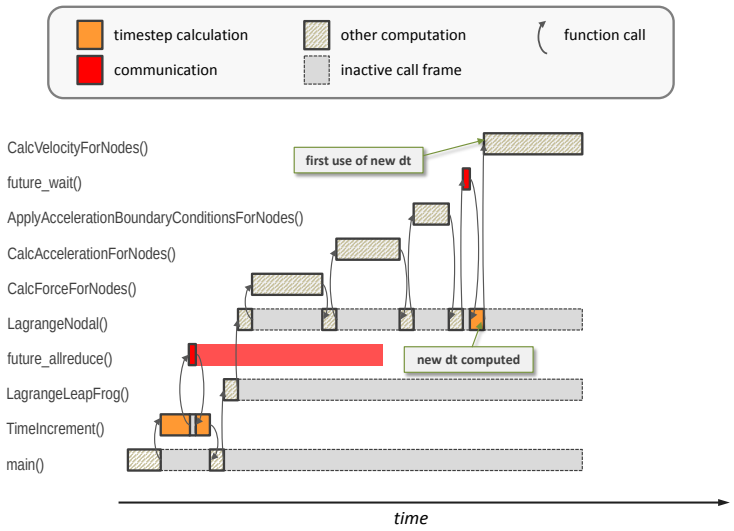
First, the two implementations differ in how they determine the time increment. Specifically, at the end of each iteration, each element computes a time increment satisfying the local Courant and Hydro constraints and the minimum value among all elements is used as the next time step. In the MPI-implementation, this is done by placing a blocking collective `MPI_Allreduce` at the beginning of each iteration (see Figure 3a). In contrast, the HPX-5 implementation replaces the `MPI_Allreduce` call with a nonblocking `future` and does not wait for its completion until after completing `ApplyAccelerationBoundaryCondition`, where the time increment is first needed (see Figure 3b). As a result, the HPX-5 implementation can effectively overlap the communication and computation phases associated with the reduction operation.

The second difference between the two versions is oversubscription. In the MPI-implementation, each core on a compute node is responsible for one domain. For the HPX-5 implementation, it is normal to assign more than one domain to one core. Oversubscription in conjunction with nonblocking synchronization semantics enable computation to overlap with communication effectively hiding network latency.

Lastly, each domain has three fixed communication patterns in the course of the computation in the HPX-5 implementation. The MPI-implementation regenerates the communication pattern with neighboring domains each time communication occurs even though it is always the same.

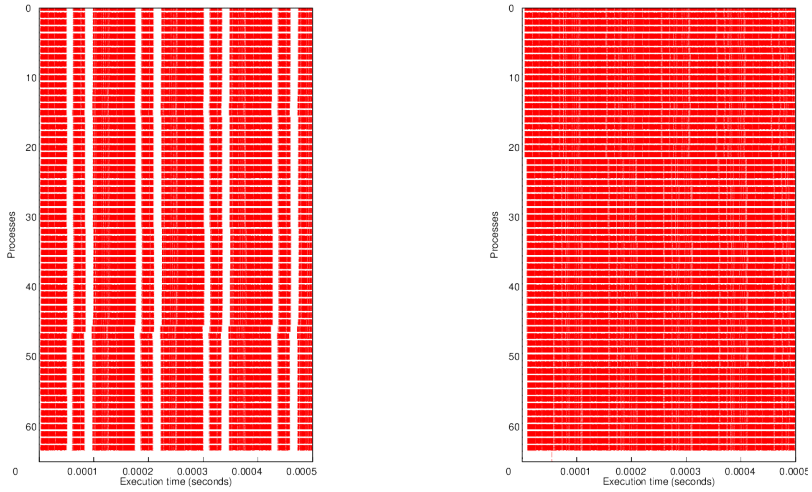


(a) Call sequence determining the global time step value along with the follow-on computations in the original MPI LULESH code. The new time step is returned by the allreduce operation which blocks until all contributing processes reach the same execution point, effectively imposing a barrier. Even though several functions could start execution (their workload does not depend on the new time step value), they are blocked until the communication phase completes.



(b) Call sequence of the HPX-5 LULESH version of the code shown above using future for a non-blocking collective operation. After `future_allreduce` returns, the pending computations that do not depend on the new time step may proceed immediately, overlapping the communication phase. A portion of the code from `TimeIncrement` function has to be moved after `future_wait` to correctly post-process the new time step value. The combined latency of `future_allreduce` and `future_wait` calls is usually substantially shorter than that of blocking allreduce on a large machine.

These changes have an immediate and visible impact on the computational phases of LULESH. Figure 4 compares the computational phases between MPI LULESH and HPX-5 LULESH on 64 processors where red indicates computation and white indicates communication. By replacing global barriers with futures based nonblocking communication, the time spent waiting for communication to complete can be reduced substantially in an application.



(a) The computational phases for MPI LULESH on 64 processors. Red indicates computation while white indicates waiting for communication.

(b) The computational phases for HPX LULESH on 64 processors. Red indicates computation while white indicates waiting for communication.

Fig. 4: A comparison of computational phases between MPI and HPX versions of LULESH.

6 Results

Strong and weak scaling results for HPX-5 LULESH are presented in this section along with the runtime system's performance predictions. All computations and simulations were performed on 16-node Xeon E5-2670 2.60 GHz based cluster with an Infiniband interconnect. The oversubscription factor for all distributed cases was two; that is, the entire LULESH computational domain was partitioned into twice as many subdomains as available cores.

Our simulation approach is most similar to SMPI [7] where online simulation (or emulation) is performed on a subset of the nodes. The rest of the nodes in the simulation are either ignored or simulated depending on the application

requirements. In case of LULESH, we computed the global values offline such that there were no message dependencies from the simulated nodes to the emulated nodes. For structured communication patterns, we use periodic boundary conditions to meet the receive depenences from the simulated nodes to the emulated nodes. Since the pending receives can generate load on the emulated nodes, we are presently working on recovering these dependences through offline traces. Communication is performed only between emulated nodes. For network simulation, we used the LogP cost model [8] to calculate communication time for the simulated nodes. Under the assumption that each parcel is sent using a single message[1], per the LogGP [3] model, a send was computed to take $(2 \times o) + (n-1)G + L$ cycles where L is the network latency, o is the overhead of transmission and G is the gap per byte. The LoGP parameters for the 16-node Xeon E5-2670 2.60 GHz based cluster were measured empirically for the above experiments.

In Figure 5, the workload was increased from 1 to 512 domains as the number of nodes were increased from 1 to 16. The simulator introduces some overhead since it has to inspect every message and either emulate or simulate it. We found that the predicted value was within 25% of the actual running time. The strong-scaling results in Figure 6 confirm the above observation. For the above runs, each "simulated" workload was run with half the number of actual nodes. Figure 7 shows the simulation accuracy of our online simulation approach. We see that the accuracy improves (that is, the difference between the emulated and simulated value decreases) as the number of emulated nodes are increased. This confirms the trade-off between simulation accuracy and the computation requirements for the simulation. As stated previously, simulating the performance of the application at Exascale levels might demand considerable computation resources. Hence, such an approach where the accuracy can be bounded by sampling a subset of the available nodes might be favorable.

7 Conclusions

Efficiency and scalability requirements for high performance computing applications has cultivated the development of new programming models which employ fine and medium grain task parallelism creating challenges for performance modeling at Exascale. In particular, task-driven approaches cause significant problems for runtime systems using lightweight concurrent threads while discrete event simulators require skeleton codes which are difficult to reliably extract from the full application codes. At the same time, runtime systems now regularly provide the introspection capability to reliably carry out performance modeling within the runtime system itself. An approach to incorporating performance modeling in the runtime system has been described here for use in cases where a prototype Exascale node is available for computation. Using a sampling approach in conjunction with a network model, a runtime system can be

[1] Almost all messages were under 32K for our HPX-5 port of the LULESH application.

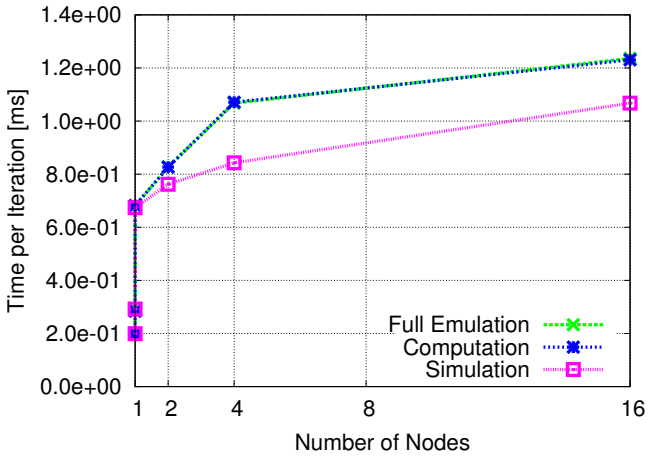


Fig. 5: Weak scaling results for HPX-5 LULESH. "Computation" represents the actual running time for a fixed workload for 500 iterations. "Full Emulation" indicates the time to perform full emulation of the workload using our hybrid emulation and simulation approach. "Simulation" shows the running time predicted by the simulator.

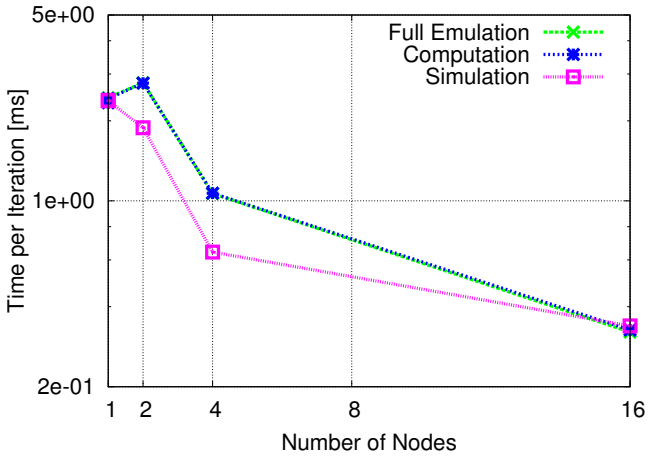


Fig. 6: Strong scaling performance of HPX-5 LULESH across 16 nodes. The description of the legend is same as the previous figure, Figure 5.

quickly transformed into a performance modeling tool without requiring traces nor discrete event simulation.

A case study has also been presented here where the LULESH proxy application has been ported to the HPX-5 runtime system and run in both of the

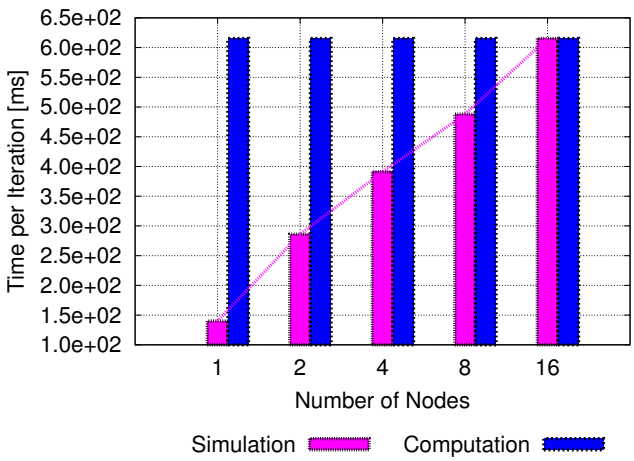


Fig. 7: Simulation accuracy as the number of emulated nodes are increased. A prediction is more accurate if the difference between the computation and simulation times is lower.

computation and simulation modalities provided by the runtime. The HPX-5 LULESH port illustrates all of the features of a many-tasking implementation, including oversubscription, asynchrony management semantics, and active messages. Strong and weak scaling results were provided for comparison between the computation and simulation modalities.

Incorporating performance modeling into modern runtime systems resolves several issues when operating at Exascale while also simplifying co-design for application developers. While such an approach is new and mostly untested, it ultimately can remove one layer of separation between application development and performance modeling for approaches employing fine and medium grain task parallelism.

8 Acknowledgments

The authors acknowledge Benjamin Martin, Jackson DeBuhr, Ezra Kissel, Luke D'Alessandro, and Martin Swany for their technical assistance.

References

1. Hydrodynamics Challenge Problem, Lawrence Livermore National Laboratory. Technical Report LLNL-TR-490254.
2. Livermore unstructured lagrangian explicit shock hydrodynamics (lulesh). Available from https://codesign.llnl.gov/lulesh.php.

3. A. Alexandrov, M. F. Ionescu, K. E. Schauser, and C. Scheiman. Loggp: incorporating long messages into the logp modelone step closer towards a realistic model for parallel computation. In *Proceedings of the seventh annual ACM symposium on Parallel algorithms and architectures*, SPAA '95, pages 95–105, New York, NY, USA, 1995. ACM.
4. M. Anderson, M. Brodowicz, A. Kulkarni, and T. Sterling. Performance modeling of gyrokinetic toroidal simulations for a many-tasking runtime system. In *Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (SC13)*, 2013.
5. L. Carrington, M. Laurenzano, and A. Tiwari. Inferring large-scale computation behavior via trace extrapolation. In *Large-Scale Parallel Processing workshop (IPDPS 2013)*, 2013.
6. L. Carrington, A. Snavely, X. Gao, and N. Wolter. A performance prediction framework for scientific applications. In *ICCS Workshop on Performance Modeling and Analysis (PMA03*, pages 926–935, 2003.
7. P.-N. Clauss, M. Stillwell, S. Genaud, F. Suter, H. Casanova, and M. Quinson. Single node on-line simulation of mpi applications with smpi. In *Parallel Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 664–675, 2011.
8. D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. Logp: towards a realistic model of parallel computation. In *Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPOPP '93, pages 1–12, New York, NY, USA, 1993. ACM.
9. G. Gao, T. Sterling, R. Stevens, M. Hereld, and W. Zhu. Parallex: A study of a new parallel computation model. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–6, 2007.
10. T. Hoefler, T. Schneider, and A. Lumsdaine. LogGOPSim - Simulating Large-Scale Applications in the LogGOPS Model. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, pages 597–604. ACM, Jun. 2010.
11. C. L. Janssen, H. Adalsteinsson, S. Cranford, J. P. Kenny, A. Pinar, D. A. Evensky, and J. Mayo. A simulator for large-scale parallel computer architectures. *IJDST*, 1(2):57–73, 2010.
12. M. Sottile, A. Dakshinamurthy, G. Hendry, and D. Dechev. Semi-automatic extraction of software skeletons for benchmarking large-scale parallel applications. In *Proceedings of the 2013 ACM SIGSIM conference on Principles of advanced discrete simulation*, SIGSIM-PADS '13, pages 1–10, New York, NY, USA, 2013. ACM.
13. K. L. Spafford and J. S. Vetter. Aspen: a domain specific language for performance modeling. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 84:1–84:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.
14. T. Sterling. Towards a Parallex-enabled Exascale Architecture. Presentation to the DOE Architecture 2 Workshop, 10 Aug 2011.
15. E. Totoni, A. Bhatele, E. Bohm, N. Jain, C. Mendes, R. Mokos, G. Zheng, and L. Kale. Simulation-based performance analysis and tuning for a two-level directly connected system. In *Proceedings of the 17th IEEE International Conference on Parallel and Distributed Systems*, December 2011.
16. G. Zheng, T. Wilmarth, O. S. Lawlor, L. V. Kalé, S. Adve, D. Padua, and P. Geubelle. Performance modeling and programming environments for petaflops

computers and the blue gene machine. In *NSF Next Generation Systems Program Workshop, 18th International Parallel and Distributed Processing Symposium(IPDPS)*, page 197, Santa Fe, New Mexico, April 2004. IEEE Press.