SAND2015-10063C

Enabling Runtime/Application Co-Design through Common Concurrency Concepts

Jeremiah J. Wilke
Sandia National Labs
Livermore, CA, 94551
jjwilke@sandia.gov

Janine C. Bennett
Sandia National Labs
Livermore, CA, 94551
jcbenne@sandia.gov

Robert L. Clay
Sandia National Labs
Livermore, CA, 94551
rlclay@sandia.gov

ABSTRACT

The best practices for advancing extreme-scale runtimes and the applications they support will likely involve co-design. Applications must change to expose as much concurrency as possible to achieve performance on future architectures, but runtimes must also adapt to address application requirements. We propose a simple classification scheme for demystifying asynchronous many-task (AMT) runtimes and apply the framework to set of runtimes and example applications. We examine possible paths forward for runtime development, and suggest how the basic concepts here can establish best practices for runtime/application co-design.

Categories and Subject Descriptors

D.3.2 [**Programming Languages**]: Language Classifications; D.1.0 [**Numerical Analysis**]: General

Keywords

Asynchronous many-task models, algorithm co-design, sparse dynamic data exchange

1. INTRODUCTION

Scaling high performance computing (HPC) performance on the path to exascale is pushing systems towards billion-way concurrency. The architectural changes to achieve this scaling will impact HPC software stacks, and consequently change both programming models and runtime systems. Applications written for petascale architectures are unlikely to automatically scale and fully exploit available parallelism on future systems. This has lead to a proliferation of task-parallel runtimes for distributed memory (Legion [2], HPX [17], X10 [11], Chapel [9], PARSEC [5], OCR [22]) and shared memory (OmpSs [6]) as well as renewed interest in long-standing systems (Uintah [15], Charm++ [21], Cilk [4]). These systems, often referred to as asynchronous many-task (AMT) runtimes, aim to maximize performance on future architectures by facilitating the expression of as much concurrency as possible, including data, task, and pipeline parallelism. Furthermore, asynchronous many-task (AMT) systems strive to manage as much of the concurrency as possible at the runtime system level, sheltering application developers from the complexities of the future architectures.

From an application developer's perspective, choosing an AMT runtime to invest in can be daunting. There are many options and the AMT community is far from a set of standards. Ultimately, AMT runtimes exist to support the applications. However, AMT concepts can inform application development, pushing application developers to refactor to new and beneficial programming or execution models. We see this balance between adapting AMT runtimes to application requirements and applications adapting to AMT concepts as the central design problem.

Long term, we believe co-design is necessary to develop AMT standards that are widely adopted. Such a co-design approach sees application, programming models, and runtime system teams collaborate to design solutions that meet the combined application and system requirements. Indeed, there is evidence that such an approach works well, as demonstrated by those runtimes that have been designed closesly with application developers from a particular scientific domain, e.g. [21, 15]. It should be noted that a runtime design decision that is useful for one application area may be a detriment for other application areas. The community lacks a comprehensive understanding regarding the interplay of runtime design decisions and performance tradeoffs across scientific application areas and system architectures. However, before we can achieve such an understanding, we argue that the AMT community must address a fundamental need, whose resolution precludes our ability to even *agree upon what the design decisions are*: common vocabularies.

In this position paper, we propose an initial classification scheme, using existing terminology where possible, to discuss how various AMT programming models and runtimes manage and express concurrency. The management of concurrency is a key aspect of a system's *execution model*, whereas the expression of concurrency is a key aspect of a system's *programming model*. Through specific examples, we classify several existing runtimes, and explore the implications of concurrency expression and management for various application areas. We believe that establishing a common vocabulary within the AMT runtime community is a critical precursor in establishing best practices, as it will facilitate co-design interactions with application areas, providing a mechanism for current AMT research efforts to compare

RESPA '15 Austin, Texas USA

and contrast their results in a more principled manner.

2. MANAGING CONCURRENCY AND HAZARDS

Basic concepts from instruction-level parallelism (ILP) naturally lend themselves to discussing the management of concurrency when coding massively parallel distributed memory systems. The distributed memory analog to ILP has been previously discussed in Legion works [2]. In ILP, out-of-order execution processors create concurrency in a serial code through dependency analysis, determining read-after-write (RAW) and write-after-read (WAR) conflicts (also known as parallel hazards). Per textbook language, RAW conflicts are true dependencies - a value cannot be read until it is written. WAR conflicts are anti-dependencies. There is no intrinsic constraint between the write and read other than a race condition. The read must complete first or it will receive the wrong value. Unlike RAW, WAR hazards can be removed. This occurs in ILP through register renaming. If a thread writes, it uses a new physical location. If a parallel thread reads, there is no longer a race condition and both threads can execute concurrently.

While these terms are already standard, we suggest their use is lacking in HPC distributed memory parallelism. In the HPC setting, concurrency can be largely understood based on how the underlying runtime manages an application's WAR and RAW conflicts. We use the term *concurrency creation* to describe how multiple threads of execution are created. Hazards exist once parallel streams of execution are consuming and producing data. We use the term *concurrency management* to describe the resolution of WAR and RAW conflicts. Although the concepts of concurrency creation and management are simple and intuitive, we argue their use provides a powerful framework for enabling runtime/application co-design, in particular facilitating a principled comparison of the effects of different AMT runtime design decisions for a particular application area. We propose the following broad categories as a useful starting point for describing concurrency creation and management:

Conservative execution: The runtime only spawns tasks in parallel that are guaranteed not to conflict. The application exposes RAW/WAR conflicts, allowing the runtime to decide which tasks can safely run in parallel. Independent threads do not need to explicitly synchronize. Execution begins with zero concurrency and grows conservatively to the maximum allowed concurrency.

Phased execution: The runtime spawns many tasks in parallel. Where RAW or WAR conflicts may exist, a phase barrier is executed to guarantee safe execution. The term phase barrier has previously been used in Legion [1] and X10 [24]. Barriers may be local operations or global collectives. Execution begins with maximum parallelism and concurrency decreases when necessary to satisfy synchronization constraints.

Copy-on-write, data-flow execution: This is an intermediate between conservative and phased execution, with the additional constraint that the application guarantees no WAR conflicts. Tasks are written to follow a write-once, read-many policy when necessary to avoid anti-dependencies. The only synchronizations required are RAW or data-flow constraints, ensuring that a value exists before a task can run. Similar to conservative execution, tasks spawn once all their data-flow constraints are met, forking new concurrency. Once running, tasks do not synchronize because there are no WAR conflicts to avoid. This approach often has higher memory requirements, and the necessary garbage collection adds complications.

Speculative execution: This represents the most distinct case and least broadly applicable. Potential parallel hazards are ignored, assuming that conflicts will usually not occur. Conflicts are detected after the fact, leading to rollback or recovery.

2.1 Categorizing Runtime Systems

Here we attempt to classify some existing systems for illustration, but omit many examples given space constraints. The classifications are broad, and each system has numerous subtleties not fully addressed here.

MPI: Message passing (MPI) generally follows phased execution with copy-on-read to remove anti-dependencies. Although threading (MPI+X) introduces new complications, the core MPI execution model is communicating sequential processes (CSPs) with each parallel worker having a private, disjoint address space. Sent messages can either pass data between processes (satisfying RAW dependencies) or they can act as phase barriers to synchronize execution. The MPI runtime provides no guarantee against WAR hazards, leaving the application to manage concurrency (particularly with non-blocking sends). A send, however, is a process-to-process copy (copy-on-read). MPI does not distinguish WAR and RAW conflicts, and a single send/recv often simultaneously resolves both a RAW and WAR dependence.

Legion: Legion's predominant execution model leverages conservative execution (although there are relaxed coherence and phase barriers for supporting many modes). Logical regions are declared as inputs to tasks with read or write permissions. Where reads and writes would conflict on a logical region, Legion performs dependency analysis to decide if two tasks can safely run in parallel. Tasks can be nested in arbitrary tree structures, but child tasks can only access a subset of the parent tasks's logical regions and permissions. With some subtleties, the Legion model is therefore fully strict execution [16]. Some support for speculative execution has been included, but is experimental [1].

Charm++: Charm++ is an actor execution model utilizing actors called *chares* [21]. Chares invoke entry methods (remotely or locally) on other chares. While the mechanics of concurrency creation are distinct from MPI (entry method calls can fork new parallelism), concurrency management is still phased execution. Although Charm++ provides the Structured Dagger [20] tool to express phases and precedence constraints, concurrency is still managed at the application level.

Concurrent Collections: Concurrent collections (CnC) [10] derives from the Linda tuple space coordination language [8], which expresses parallelism via puts and gets from a key-value store. Data blocks must receive unique string labels, ensuring idempotent, data-flow

execution. While this adds the minor programming burden of expressing unique keys for every data block, concurrency is not managed at the application level. The runtime ensures RAW (data-flow) dependences are satisfied for each task and WAR dependences are completely avoided through the write-once policy.

3. EXPRESSING CONCURRENCY

The creation and management of concurrency within the runtime impacts the expression of concurrency via a system's programming model. Phased execution is highly flexible, but forces the application to manage concurrency, creating a significant programming burden in the application. The application, not the runtime, must express phase barriers and ensure they correctly avoid RAW and WAR conflicts. If coarse-grained phase barriers are used, they might also over-express synchronization.

Copy-on-write tasks simplify concurrency management, but may induce performance overheads. Given an appropriate programming model (e.g. CnC), execution is straightforward to express. Each data block, when written, must be given a unique identifier. Many science and engineering applications are structured around iterations, creating a natural naming scheme, however this is typically at the cost of creating extra storage or data movement.

Conservative execution can push concurrency management into the runtime system, and avoids the performance overheads associated with copy-on-write execution. Rather than managing concurrency in the application, the runtime manages parallel tasks. If the runtime system supports a data model like OmpSs or Legion, the application only needs to express a data-centric description of the algorithm and the runtime can even derive RAW and WAR conflicts. Conservative execution, though, creates different difficulties at the application level. All inputs and outputs to a task must be declared before the task begins executing. Once executing, a task cannot write or read any data it has not *a priori* requested permissions for. Thus data-dependent, dynamic applications (e.g. Section 4.2) may be more naturally expressed via another concurrency management scheme since tasks may not know all required inputs until after beginning execution.

4. APPLICATIONS

We consider four science and engineering domains to illustrate each of the execution categories.

- Direct numerical simulation (DNS) of complex combustion phenomena (S3D)
- Particle-in-cell with charged particle migrations
- Fail-stop fault recovery
- Discrete event simulation

4.1 S3D

S3D [12] is an excellent candidate for MPI+X parallelism, in which distributed and thread-level concurrency are managed separately. A regular 3-dimensional mesh is partitioned across parallel ranks, and each mesh point contains many fields for the physical quantities associated with different chemical species. At each timestep, ranks compute finite difference stencils, which requires ghost data to be exchanged between ranks. The top-level SPMD partitioning of the problem does not map well to conservative execution. Data is not disjoint across parallel ranks, and the overlapping ghost regions therefore cause conflicts between all neighbors. In this use case, managing the SPMD parallelism in the application (as opposed to runtime) is relatively straightforward. The most natural concurrency mode in this setting is phased execution with, in S3D's various implementations, MPI_Send/Recv calls or region-to-region copies in Legion acting as phase barriers.

Although the top-level SPMD parallelism is most naturally handled within the application, once ghost data is exchanged, abundant on-node parallelism can be exploited without further MPI or distributed memory communication. S3D is a regular application and all task inputs are known *a priori*, The Legion S3D implementation exposes and manages additional concurrency [3] by allowing different fields on the same mesh point to be "sliced" and passed independently to tasks. The complex S3D task graph in Figure 1 (for even a simple mechanism) highlights the benefits as the complexity of the task-graph would make application-level management very burdensome. Where necessary, the application explicitly manages parallelism, but then passes off concurrency management to the Legion runtime to handle the task graph in Figure 1.

4.2 Particle-in-Cell

In particle-in-cell (PIC) codes [13], charged particles migrate, interacting with electric and magnetic fields and even potentially colliding to create more particles. Particle migration is split into macro- and micro- time steps. Fields are updated every macro time step while particle migration is usually tracked at finer scales (micro steps) to record charge deposition for updating electric fields. Unlike S3D, a PIC micro-iteration cannot know *a priori* how many particles will migrate outward (sent to other mesh regions) or inward (received from other mesh regions). A macro-iteration can also have an arbitrary number of micro-steps, only finishing once all mesh regions are done moving particles.

A natural and efficient expression of a PIC implementation sees parallel workers asynchronously push particles around the system, responding to new incoming particles as they are available. Only a single, global phase barrier is required to ensure quiescence or termination detection of migrating particles. It is less clear how to efficiently express and efficiently implement a PIC code with runtimes requiring *a priori* task input declaration. A task could express the maximum number of particles to migrate along with a maximum number of send/recv partners. However, this creates inefficiencies in the algorithm as over-expressing task inputs creates false and unnecessary dependencies.

4.3 Fault Tolerance

Fault tolerance is an extreme case of task dependency that cannot be known *a priori*. Checkpoints are essentially phase barriers. During rollback recovery, a process reads data from previous steps. Fault recovery is therefore technically a WAR dependence. A parallel worker cannot overwrite its old data because at some point in the future a failure might occur, requiring the failed process to restart and re-read the data. Data is therefore written to persistent storage (copy-on-write), removing the anti-dependence.

While checkpointing copies state at regular phase barriers, a more extreme resilience model would require all tasks to be write-once. Tasks never overwrite data from previous iter-

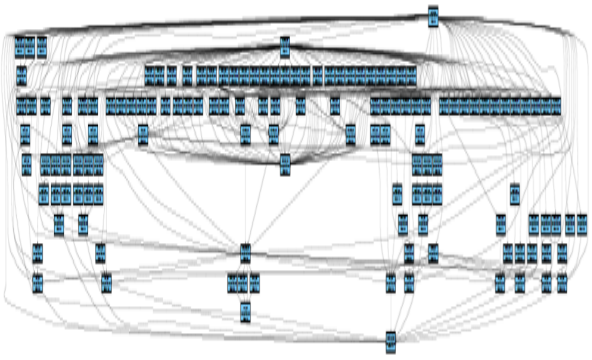

Figure 1: Task graph generated by Legion for simple combustion mechanism showing the abundant task-level parallelism in S3D, image courtesy of Dr. Jacqueline Chen and the Legion team.

ations, instead migrating them to cheap, persistent storage. The scheme completely avoids WAR conflicts, maximizing concurrency during recovery at the cost of extra storage.

4.4 Discrete Event Simulation

Parallel discrete event simulation (PDES) [14] represents a dramatic shift from previous examples. Discrete actors generate and receive events, advancing time after each event. Discrete event simulations feature prominently in network simulations [23], simulating packets as events on network routers. Discrete events carries the implicit dependency of global virtual time to all tasks. Each actor maintains its own local clock, but events must globally preserve time ordering. If discrete actors run in parallel, one actor's local time may lag behind. If actors exchange events with different local times, events may schedule in the past, violating time order.

PDES is similar to PIC in that parallel actors cannot know *a priori* what events they will receive. Conservative PDES methods use safe lookahead values to create parallelism, ensuring that no time violations occur. These methods require constant phase barriers (either point-to-point messages or global reduces) to ensure parallel actors are synchronized. These phase barriers, however, can potentially represent false dependencies and limit concurrency. Optimistic PDES protocols use speculative execution, assuming that time violations will not occur. When time violations do occur, a rollback method is invoked to correct the state. This optimistic protocol features prominently in the ROSS simulator [7], which uses the reverse computation ideas from the Time Warp algorithm [19] to correct mis-speculation.

5. CALL FOR DEVELOPING BEST PRACTICES BY REQUIREMENTS-DRIVEN CO-DESIGN

In this position paper we illustrate how the use of simple and intuitive concepts such as concurrency creation and management can facilitate a principled comparison of the effects of different AMT runtime design decisions for different application areas. We believe that the development of such common vocabularies is a necessary precursor to the development of best practices and eventual standards. In the short discussion presented here, we see how the use of such terms facilitates the exploration of the design decision trade-off space. While conservative execution enables appealing data-centric programming models and automatic concurrency management, it may complicate the expression of sparse or dynamic data exchange workflows [18]. In the same way, phased execution can express highly dynamic applications, but it puts the burden of concurrency management at the application level.

We suggest the critical design issue facing runtime development is choosing between 1) a single execution style for the runtime and forcing applications to adapt, 2) forcing a runtime to accommodate several execution styles suited to many applications, or 3) developing several runtimes optimized for different application workloads. Deciding which option above is the most sustainable can only be achieved by a concerted co-design effort between application, programming model, and runtime developers centered on common concepts and vocabulary for discussing requirements.

6. ACKNOWLEDGMENTS

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000. This work was supported by the U.S. Department of Energy (DOE) National Nuclear Security Administration (NNSA) Advanced Simulation and Computing (ASC) program. Special thanks to Sean Treichler for useful discussion and use of figures.

7. REFERENCES

[1] M. Bauer. *Legion: Programming Distributed Heterogeneous Architectures with Logical Regions*. PhD thesis, Stanford University, 2014.

[2] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken. Legion: expressing locality and independence with logical regions. In *SC '12: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11, 2012.

[3] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken. Structure slicing: Extending logical regions with fields. In *Supercomputing (SC)*, 2014.

[4] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An Efficient Multithreaded Runtime System. *SIGPLAN Notices*, 30:207–216, 1995.

[5] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, T. Herault, and J. J. Dongarra. PaRSEC: Exploiting Heterogeneity to Enhance Scalability. *Comput. Sci. & Engineering*, 15:36–45, 2013.

[6] J. Bueno, J. Planas, A. Duran, R. M. Badia, X. Martorell, E. Ayguade, and J. Labarta. Productive Programming of GPU Clusters with OmpSs. In *IPDPS: 26th International Parallel & Distributed Processing Symposium*, pages 557–568, 2012.

[7] C. D. Carothers, D. Bauer, and S. Pearce. ROSS: A High-Performance, Low Memory, Modular Time Warp System. In *PADS 2000: 14th Workshop on Parallel and Distributed Simulation*, pages 53–60, 2000.

[8] N. J. Carriero, D. Gelernter, T. G. Mattson, and A. H. Sherman. The Linda Alternative to Message-Passing Systems. *Parallel Comput.*, 20:633–655, 1994.

[9] B. Chamberlain, D. Callahan, and H. Zima. Parallel Programmability and the Chapel Language. *Int. J. High Perform. Comput. Appl.*, 21:291–312, 2007.

[10] A. Chandramowlishwaran, K. Knobe, and R. Vuduc. Performance Evaluation of Concurrent Collections on High-Performance Multicore Computing Systems. In *IPDPS 2010: 24th International Parallel and Distributed Processing Symposium*, pages 1–12, 2010.

[11] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. V. Praun, and V. Sarkar. X10: An Object-Oriented Approach to Non-Uniform Cluster Computing. In *OOPSLA 2005: 20th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 519–538, 2005.

[12] J. H. Chen, A. Choudhary, B. de Supinski, M. DeVries, E. R. Hawkes, S. Klasky, W. K. Liao, K. L. Ma, J. Mellor-Crummey, N. Podhorszki, R. Sankaran, S. Shende, and C. S. Yoo. Terascale direct numerical simulations of turbulent combustion using S3D. *Computational Science and Discovery*, page 015001, 2009.

[13] V. K. Decyk and T. V. Singh. Particle-in-cell algorithms for emerging computer architectures. *Computer Physics Communications*, 185(3):708 – 719, 2014.

[14] R. M. Fujimoto. Parallel Discrete Event Simulation. *Commun. ACM*, 33:30–53, 1990.

[15] J. D. D. S. Germain, S. G. Parker, C. R. Johnson, and J. McCorquodale. Uintah: a massively parallel problem solving environment. 2000.

[16] Y. Guo, R. Barik, R. Raman, and V. Sarkar. Work-first and help-first scheduling policies for async-finish task parallelism. In *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–12, May 2009.

[17] T. Heller, H. Kaiser, and K. Iglberger. Application of the ParalleX execution model to stencil-based problems. *Comput. Sci.*, 28:253–261, 2013.

[18] T. Hoefler, C. Siebert, and A. Lumsdaine. Scalable communication protocols for dynamic sparse data exchange. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 159–168, 2010.

[19] D. Jefferson and H. Sowizral. *Fast Concurrent Simulation using the Time Warp Mechanism: Part I*. 1982.

[20] L. Kalè and M. Bhandarkar. Structured dagger: A coordination language for Message-Driven Programming. In *Euro-Par'96 Parallel Processing*. Springer Berlin Heidelberg, 1996.

[21] L. V. Kale and S. Krishnan. CHARM++: A Portable Concurrent Object Oriented System Based on C++. In *OOPSLA 1993: 8th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 91–108, 1993.

[22] T. Mattson, R. Cledat, Z. Budimlic, V. Cave, S. Chatterjee, B. Seshasayee, R. van der Wijngaart, and V. Sarkar. OCR: The Open Community Runtime Interface. Technical report, June 2015.

[23] A. F. Rodrigues, K. S. Hemmert, B. W. Barrett, C. Kersey, R. Oldfield, M. Weston, R. Risen, J. Cook, P. Rosenfeld, E. CooperBalls, and B. Jacob. The Structural Simulation Toolkit. *ACM SIGMETRICS Perform. Eval. Rev.*, 38:37–42, 2011.

[24] J. Shirako, D. M. Peixotto, V. Sarkar, and W. N. Scherer. Phasers: A unified deadlock-free construct for collective and point-to-point synchronization. In *Proceedings of the 22Nd Annual International Conference on Supercomputing*, ICS '08, pages 277–288, New York, NY, USA, 2008. ACM.