Long-time dynamics through parallel trajectory splicing

Danny Perez, Ekin Dogus Cubuk, Amos Waterland, Efthimios Kaxiras, and Arthur F. Voter

J. Chem. Theory Comput., Just Accepted Manuscript • DOI: 10.1021/acs.jctc.5b00916 • Publication Date (Web): 24 Nov 2015
Downloaded from http://pubs.acs.org on November 30, 2015

Just Accepted

“Just Accepted” manuscripts have been peer-reviewed and accepted for publication. They are posted online prior to technical editing, formatting for publication and author proofing. The American Chemical Society provides “Just Accepted” as a free service to the research community to expedite the dissemination of scientific material as soon as possible after acceptance. “Just Accepted” manuscripts appear in full in PDF format accompanied by an HTML abstract. “Just Accepted” manuscripts have been fully peer reviewed, but should not be considered the official version of record. They are accessible to all readers and citable by the Digital Object Identifier (DOI®). “Just Accepted” is an optional service offered to authors. Therefore, the “Just Accepted” Web site may not include all articles that will be published in the journal. After a manuscript is technically edited and formatted, it will be removed from the “Just Accepted” Web site and published as an ASAP article. Note that technical editing may introduce minor changes to the manuscript text and/or graphics which could affect content, and all legal disclaimers and ethical guidelines that apply to the journal pertain. ACS cannot be held responsible for errors or consequences arising from the use of information contained in these “Just Accepted” manuscripts.
Long-time dynamics through parallel trajectory splicing

Danny Perez\(^1\), Ekin D. Cubuk\(^2\), Amos Waterland\(^2\), Efthimios Kaxiras\(^2\#\), and Arthur F. Voter\(^1\%\)

1. Theoretical Division T-1, Los Alamos National Laboratory, PO Box 1663, Los Alamos, NM, 87544
2. Department of Physics and School of Engineering and Applied Sciences, Harvard University, Cambridge, Massachusetts 02138, United States

*email: danny_perez@lanl.gov
#email: kaxiras@physics.harvard.edu
%email: afv@lanl.gov

Abstract

Simulating the atomistic evolution of materials over long timescales is a longstanding challenge, especially for complex systems where the distribution of barrier heights is very heterogeneous. Such systems are difficult to investigate using conventional long-timescale techniques and the fact that they tend to remain trapped in small regions of configuration space for extended periods of time strongly limits the physical insights gained from short simulations. We introduce a novel simulation technique, Parallel Trajectory Splicing (ParSplice), that aims at addressing this problem through the timewise parallelization of long trajectories. The computational efficiency of ParSplice stems from a speculation strategy whereby predictions of the future evolution of the system are leveraged to increase the amount of work that can be concurrently performed at any one time, hence improving the scalability of the method. ParSplice is also able to accurately account for, and potentially reuse, a substantial fraction of the computational work invested in the simulation. We validate the method on a simple Ag surface system and demonstrate substantial increases in efficiency compared to previous methods. We then demonstrate the power of ParSplice through the study of topology changes in Ag\(_{42}\)Cu\(_{13}\) core-shell nanoparticles.

1. Introduction

Atomistic simulations occupy a central role in materials science, chemistry, and biology. They are often essential in predicting the properties of materials, motivating theories, and interpreting experiments. In that context, molecular dynamics (MD) is widely considered as the gold standard, as it is essentially exact (for a given form of the interatomic interactions). However, MD is notoriously limited in terms of timescales, because integrating equations of motion is inherently sequential: work can proceed on only one (or maybe a few\(^1\)) timesteps at a time. While this task can be parallelized efficiently for large systems, direct approaches are ill-suited for simulating long time-scales, which is a particular problem when the dynamics are activated,
i.e., consisting of long periods of uneventful vibrations punctuated by rapid jumps to a new configuration. In that case, the timescale limitations of conventional methods often imply that only a few transitions, or perhaps none, will be observed on MD timescales. This is an unfortunate state of affairs, especially given the increasingly widespread availability of massively parallel computers.

When dynamics involve activated processes, the infrequent nature of the dynamics can be leveraged to accelerate them. Indeed, rare events are “simple” because they can be described in statistical terms, e.g., using rate theories. This property has been exploited over the last two decades by so-called Accelerated Molecular Dynamics (AMD) methods. AMD is here used in its original sense denoting MD-based methods for the generation of dynamically correct state-to-state trajectories, not as describing enhanced-sampling techniques, which were based on the aforementioned methods.) The spirit of AMD is to use MD as a computational engine to generate statistically correct state-to-state trajectories without introducing assumptions on the nature of the possible transitions, using statistical concepts to coax it into doing so faster. In light of the above discussion, one of the AMD methods --- Parallel Replica Dynamics (ParRep) --- has been the subject of particular attention over the last few years. As the name suggests, ParRep harnesses the power of large-scale parallel computers by parallelizing the state-to-state trajectory in the time domain, therefore enabling the long-time simulations of small systems. Since their inception, AMD methods, which also include Temperature Accelerated Dynamics (TAD) and hyperdynamics, have proven able to significantly extend the reach of atomistic simulations for a wide range of systems while maintaining accuracy close to that of MD. However, certain classes of systems remain difficult to simulate by either MD or AMD methods. In long-time simulation jargon, this is often referred to as the “low-barrier problem”, but it might be more accurately described as the “heterogeneous barrier” problem. Indeed, when all barriers are low, standard MD might be sufficient while, when all barriers are high, AMD methods are adept at accelerating the dynamics. If, however, the system contains both high and low barriers, and the physics of interest involves crossing the high barriers, both methods typically fail. Consider the common case where the energy landscape contains super-basins, where sets of states are internally connected by fast pathways, but where inter-set transitions are slow, e.g., a defect that can readily reorganize locally into a number of conformations, but that diffuses only slowly. In that case, a trajectory will remain in the same set of states for a long time before leaving. This is a difficult situation as the amount of acceleration available from AMD is limited by the fastest processes, so that escape from the super-basin might be out of reach. Note that this problem is not specific to AMD, but also plagues kinetic Monte Carlo (KMC) approaches.

In the following, we introduce Parallel Trajectory Splicing (ParSplice), a generalization of ParRep that addresses the low-barrier problem. The key idea is that long trajectories can be decomposed into “segments” that can be concurrently generated in many different states. These segments can then be assembled, or spliced, into longer trajectories. Based on a powerful and very general formal decomposition of the trajectory into independent segments, we show that ParSplice can generate arbitrarily accurate dynamics. Through the introduction of speculative trajectory segment generation, it is further shown to be ideally suited to exploit massively parallel computers, even for systems whose heterogeneous distributions of barriers...
lead to the formation of super-basins. ParSplice can hence dramatically increase the range of systems amenable to very long time, high quality, simulation. A similar idea to ParSplice has been investigated in a different setting, to speed up and parallelize general purpose computation \(^{9,10}\).

As with the other AMD methods, ParSplice aims at generating an individual, dynamically correct, trajectory that spans very long timescales. In that respect, it essentially aims at emulating direct MD, in that, from a given starting point, the trajectory evolves in a completely unbiased manner. This distinguishes ParSplice from other long-timescale methods that also rely on the parallel generation of trajectory segments. For example, milestoneing \(^{11}\), forward flux sampling \(^{12}\), and transition interface sampling \(^{13}\) are double-ended methods that target specific transitions (e.g., folding of a protein or melting of a crystal) through the choice of a progress variable. Their goal is to either estimate the transition rate for that particular transition or to sample from the ensemble of reactive trajectories connecting both end-points. Markov State Models (MSM) \(^{14}\) is another class of methods that can harvest large numbers of short trajectories generated in parallel. MSM use extensive sampling in the form of a large number of short trajectories to approximate a few low-lying eigenvalues and eigenvectors of the generator of the dynamics. From these, once can identify and characterize a few of the dominant kinetic bottlenecks in the systems. Each class of method therefore excels for different problems: when one is interested in a specific transition for which a reasonable reaction coordinate is available, double-ended methods are a good choice. If one is instead interested in discovering and characterizing a few dominant kinetic bottlenecks (corresponding to the very slowest relaxation modes of the system) and when extensive sampling is affordable, MSM models can be extremely powerful. Finally, when the evolution of the system is complex and a priori difficult to characterize in terms of progress variables or reaction coordinates, when extensive sampling is prohibitive because of the complexity of the system, or full dynamical details are needed, MD-like methods like ParSplice are preferred.

The manuscript is organized as follows: in Section 2, we first describe the trajectory splicing procedure and analyze its theoretical underpinnings; we then describe, in Section 3, how trajectory splicing ideas can be leveraged in practice to form the basis of a parallel method, ParSplice; Section 4 reports on a validation of the method through an investigation of the dynamics of a silver trimer evolving on a silver (100) surface; in Section 5, we then use ParSplice to investigate the evolution of a silver/copper core-shell nanocluster on timescales approaching a millisecond, before concluding.

2. Trajectory Splicing

Perhaps the easiest way to introduce trajectory splicing is by considering a continuous-time Markov chain on a discrete state space. Such a chain is specified by the set of transition rates \(k_{ij}\) between any pair of states \(i\) and \(j\). Dynamics on this state-space are Markovian: the probability of jumping to state \(j\) in the next infinitesimal time increment is independent of time and of the past history. Now consider the following thought experiment. Using all possible states
as starting points, generate a number of trajectory segments using a conventional KMC algorithm and store the final states $m$ and corresponding trajectory durations $\tau$ in a database. Now generate a long trajectory by the following procedure: starting from a state $l$, select from the database the first trajectory segment that starts in $l$; increment the simulation time by the segment duration $\tau$ and move the current state to the corresponding final state $m$; repeat until running out of segments. It is clear that such a trajectory would be statistically indistinguishable from a trajectory generated from scratch, as all segments starting in a given state are statistically equivalent. This argument also applies if the database stores segments that are short (for which the initial and final states are the same), long (containing multiple transitions), or even random length, as long as each one of them is generated in an unbiased fashion. The generation of such a trajectory can thus be parallelized on a very large number of processors.

![Figure 1: Illustration of the concept of spliceable segment. The red star marks the beginning of an MD run in state 1. The dashed lines represent state boundaries. Any section of this trajectory that begins and ends in blue regions is a spliceable segment. Such segments however cannot contain any portion of the red section, or begin or end in yellow sections.](image)

This concept can be extended, with controllable error, to cases where the dynamics are not Markovian. The derivation is inspired by conventional concepts of separation of timescales in kinetics and specifically relies upon the treatments given in Refs. 6,17 for ParRep. The derivation, reproduced for completeness, is here reinterpreted as a means to achieve trajectory splicing. As in these references, we will consider the case of overdamped dynamics evolving on a continuous potential. The same arguments can be generalized to other cases where quasi-stationary distributions exist, such as Langevin dynamics. Generalization to microcanonical MD might be possible, but would probably entail additional conditions on the potential. Setting the friction coefficient and the masses to unity without loss of generality, the equations of motion are given by:

$$dX_t = -\nabla V(X_t)dt + \sqrt{2/\beta}dW_t,$$

where $X_t$ is a point in configuration space, $V$ is the interatomic potential, $dW_t$ is a delta-correlated stationary Gaussian process with zero mean and unit variance. We define states as the region of configuration space enclosed in a connected hyper-surface $\Omega$ (i.e., all pairs of points inside a state are connected by paths that are entirely contained within the state). The union of all states is assumed to cover the whole configuration space, but this assumption can
be relaxed. Now consider a trajectory evolving on that space. We define a \textit{spliceable trajectory segment} (or simply a segment) as a section of trajectory such that i) \textit{the trajectory spent at least a time} $\tau_c$ (the so-called correlation time) \textit{in a single state immediately prior to the beginning of the segment}, and ii) \textit{the trajectory spent at least a time} $\tau_c$ \textit{within a single state} (potentially, but not necessarily, a different one from that at the beginning of the segment) \textit{immediately prior to the end of the segment}. This condition is illustrated in Fig. 1. As a consequence of this definition, portions of trajectories that remain for less than $\tau_c$ within a given state can only be found in the interior of a segment, not at the beginning or end. We now show that segments defined in this way can be spliced into longer trajectories just as we can splice Markovian segments.

Consider a point $X_0$ in configuration space located in state $\alpha$. At a later time $t$, conditional on not having escaped $\alpha$ during that time, the final configurations are distributed according to $P(X, t) = P(X, t | X_0)$, where $P(X, t)$ obeys the Smoluchowski equation

$$\partial_t P = LP$$

in $\alpha$ with boundary condition $P = 0$ on $\Omega_\alpha$, where $L = -\nabla \cdot (\nabla \cdot ) + \beta^{-1} \nabla^2$. Using a spectral decomposition of $L$ in terms of its negated eigenvalues $\lambda_i > 0$ (i.e., the eigenvalues of $L$ are $-\lambda_i$), assumed to be sorted in increasing order, and the corresponding eigenfunctions $u_i(X)$, $P(X, t)$ can be written as

$$P(X, t) = \sum_i c_i^0 \exp(-\lambda_i^\alpha t) u_i^\alpha(X),$$

where the $c_i^0$ are constants such that $P(X, 0) = \delta(X_0)$; i.e. $c_i^0 = \int_{\Omega_\alpha} \delta(X_0) u_i^\alpha(X) d\mu^{-1}(X)$, where $\mu(X)$ is the invariant measure of the original dynamics (the Boltzmann distribution). At long times, $t \gg 1/(\lambda_2^\alpha - \lambda_1^\alpha)$, the distribution simplifies to

$$P(X, t) \approx c_1^0 \exp(-\lambda_1^\alpha t) u_1^\alpha(X).$$

As is now apparent, the normalized probability distribution in the long-time limit, conditional on having remained in the same state, is given by $u_1^\alpha(X)$ and is invariant with time. For this reason, $u_1^\alpha(X)$ is referred to as the Quasi-Stationary Distribution (QSD) for state $\alpha$. It can be shown that, starting from the QSD, the first-escape dynamics from $\alpha$ is Markovian, i.e., the probability per unit time that the trajectory escapes through a given surface element $d\Omega_\alpha$ is independent of time and the escape location is uncorrelated with the escape time $\tau_c$. This simple fact has profound consequences for trajectory splicing: in a statistical sense, because they are both samples from the QSD, the \textit{end point of a trajectory that just spent more than $\tau_c$ in $\alpha$ is statistically equivalent to the beginning of a trajectory that just spent more than $\tau_c$ in $\alpha$ w.r.t. predicting the next escape pathway and time, with an error of order $\exp[-(\lambda_2^\alpha - \lambda_1^\alpha)\tau_c]$}. In other words, generating a trajectory by joining spliceable segments (as defined above) gives statistically correct results with an error that scales as $\exp[-(\lambda_2^\alpha - \lambda_1^\alpha)\tau_c]$. As $\tau_c$ is an adjustable parameter, a spliced trajectory can be made arbitrarily accurate. The error term stems from the fact that, formally, the system only exponentially approaches the QSD but never truly reaches it. The Markovian limit is taken as $\lambda_2 \to \infty$, in which case the
decomposition is statistically exact even for $\tau_c = 0$; i.e., a Markovian trajectory can be arbitrarily partitioned into segments.

This derivation provides an extremely convenient and rigorous representation of long-time dynamics in terms of periods where complex correlated transitions occur, and periods where a simple representation in terms of Markovian behavior is appropriate. In the former case, corresponding to periods where the trajectory contains a transition between different states, a fully-resolved, continuous, and dynamically accurate representation is required in order to describe complex dynamical behavior (recrossing of the dividing surfaces between states, multiple, correlated, transitions in short succession). In the latter case, where the trajectory remains trapped in the same state for an extended period of time, statistical tools, such as the QSD, can capture the statistics of next-escapes to an arbitrary accuracy. In that sense, it bears resemblance to the dynamically-corrected transition state theory (TST) of Chandler and to its multistate generalization by Voter and Doll through which classically exact rates can be derived by combining a traditional TST representation in terms of first-order kinetics and corrections obtained by dynamically generating short trajectories. In the current context, the TST rate is replaced by the escape rate from the QSD in each state, and the dynamical correction factor is the probability that the endpoint of a trajectory sampled from the QSD of $\alpha$ first relaxes to the QSD of $\beta$ (i.e., that $\beta$ is the first state it spends at least $\tau_c$ in) after having left $\alpha$. The result is also akin to a dynamically corrected Markov State Model, where arbitrary accuracy could be achieved for any partition of configuration space into states. As will be shown below, combining these two levels of representation allows for efficient and accurate numerical techniques.

We now show how this concept can be leveraged in the context of AMD to significantly extend the timescale amenable to direct simulation through the time-wise parallelization of the trajectory.

3. Parallel Trajectory Splicing

A trajectory generated by a single MD instance can be decomposed into segments; while this might be instructive, there is no computational gain (or loss) in doing so. The real potential of trajectory splicing lies in the concurrent generation of segments in many states from many MD instances. This is the essence of ParSplice. Note that all "interesting" sections of the trajectory, i.e., when correlated transitions occur, are fully resolved and fully continuous in this scheme. Continuity of the trajectory in configuration space is however lost when splicing segments, as the end points of spliceable segments starting and ending in the same state are extremely unlikely to correspond to precisely the same point in configuration space. However, such discontinuities only occur when the trajectory remains trapped in a given state for so long that its precise configuration does not contain information that pertains to the longer time kinetics beyond the fact that it is a sample from the QSD. In other words, these discontinuities are such that they entail no loss of kinetically relevant information as far as long-time dynamics are concerned.
While different implementations of ParSplice are possible, we focus here on a particularly simple one for clarity of exposition. We consider an online version of ParSplice where the database of segments is dynamically populated as the simulation proceeds. A basic ParSplice implementation contains two types of processes: a splicer, and \( N \) producers. As shown in Algorithm 1, producers fill out requests for segments beginning in a given state and return the results. The only requirement for producers is that they return segments that are unbiased samples from the ensemble of spliceable segments starting in a given state. This can be achieved by using a simple rule where a segment is returned as soon as it is longer than a given length \( \tau_s \) and is spliceable. While different rules are conceivable, they should be agnostic to the physics along the trajectory, e.g., one should not decide to backtrack to find an earlier spliceable endpoint because the trajectory began a long sequence of correlated transitions, as this could bias the spliced trajectory in favor of segments that do not contain transitions.

```
• Until end of simulation:
  o Request initial state from the splicer
  o Set \( v=0 \) (duration of the latest visit to a state)
  o #Prepare a spliceable starting point in the initial state
  o While \( v < \tau_c \):
    • Run MD for a time \( t \)
    • \( v+=t \)
    • Check if transition during last \( t \)
    • If transition:
      • \( v=0 \)
      • reload the initial state
  o #Generate a spliceable segment
  o Set \( d=0 \) (duration of the whole segment), \( v=0 \) (duration of the latest visit to a state)
  o While \( v < \tau_c \) or \( d < \tau_s \):
    • Run MD for a time \( t \)
    • \( d+=t \)
    • Check if transition during last \( t \)
    • If transition:
      • \( v=0 \)
  o Return \( d \) and final state to the splicer
```

Algorithm 1: Producer algorithm

As shown in Algorithm 2, the splicer is more complex: it manages the database of segments, generates a trajectory by consuming segments from the database, and schedules the production of additional segments in specific states. The database is implemented as a set of lists of segments, each grouped by their respective initial state. The database contains two types of segments: incomplete segments, which correspond to tasks that have been scheduled but not yet fully executed, so that only the initial state is known, and completed segments, which are segments that have been fully processed by a producer and for which both end-points are available. As discussed below, new (incomplete) segments are appended to the end of the
respective lists but (completed) segments are consumed by the splicer from the beginning, in a first-in-first-out fashion. The splicing procedure itself is very simple: one repeatedly assesses whether a completed segment is available at the beginning of the list of segments corresponding to the current end state of the trajectory; if so, that segment is deleted from the database, spliced to the end of the trajectory, and the whole procedure repeated; if not, the procedure returns. Note that looking for completed segments only at the beginning of the corresponding list (in contrast to picking any available completed segment at random) is crucial to enforce correctness. This is because there is, in general, a correlation between the characteristics of a segment and the time it takes to generate it. For example, completing a segment that contains a transition requires running at least $\tau_c$ beyond the transition point, so that, on average, segments containing transitions take longer to generate. Without imposing an ordering (i.e., that the first segment to be scheduled in a given state should be consumed first), segments that take a shorter time to generate would have an artificially high probability of becoming part of the trajectory, potentially introducing a bias by delaying the incorporation of segments that contain transitions. Of course, there are other ways to guard against such bias; for example one could also randomly pick from any kind of segment but wait for its completion in case the selected segment is incomplete. Note that this kind of procedure is not required for an offline implementation where all segments are generated before the splicer starts consuming from the database; in-order or random sampling from all available segments with the right starting point would both be proper.

- Until end of run:
  - Receive a completed segment from a producer
  - Mark the corresponding segment in the database as completed
  - Extend the spliced trajectory as much as possible
  - Set $c$=current end of the spliced trajectory
  - While database[c].first is a completed segment:
    - Remove database[c].first from the database and append it to the end of the spliced trajectory
    - Set $c$=current end of the spliced trajectory
  - Schedule a new segment for the producer
  - m=schedule()
  - Send the new starting state $m$ to the producer
  - database[m].append(incomplete segment)

Algorithm 2: Splicer algorithm

Last, but not least, ParSplice requires a procedure to schedule the production of new segments to be added to the database, i.e., to decide in which state the next segment should start (schedule() in Algorithm 2). This procedure is key to the ability of the method to efficiently exploit very large numbers of processors and to handle the low-barrier problem. Two possibilities are shown in Algorithm 3. The most straightforward approach, the so-called “actual end” approach, is simply to schedule segments that start at the current end of the trajectory. In that case, ParSplice is roughly equivalent to an implementation of ParRep that would leverage a database to store and reuse all of the information generated by the replicas. While already a notable improvement, this choice is not optimal. Consider a case where an estimation of the typical escape time from the current end state is available. Looking at the number of segments that
have already been scheduled but are not yet completed might reveal that it is in fact unlikely that additional segments will be needed to let the system escape. If, moreover, one has some knowledge of the most likely states to be visited next, it might then be more advantageous to schedule segments in one of these states instead. This way, when (if) the trajectory reaches that state in the future, segments might already be available, or at least, underway. This element of speculation is key to expose additional parallelization opportunities, and hence to efficiently exploit a large number of processors.

Algorithm 3: Scheduler algorithm

To this end, we propose a simple solution: segments are requested at the end of virtual trajectories. Virtual trajectories are equivalent to usual spliced trajectories except that, if an incomplete segment is encountered during splicing, its endpoint is statistically sampled. We use a Maximum Likelihood estimator for the probability that the final state of a segment that starts in \( \alpha \) is \( \beta \), i.e., \( p_{\alpha \rightarrow \beta} = \frac{N_{\alpha \rightarrow \beta}}{N_{\alpha \rightarrow \cdot}} \), where \( N_{\alpha \rightarrow \cdot} \) is the total number of completed segments that were generated in state \( \alpha \) (including those which have already been consumed by the splicer) and \( N_{\alpha \rightarrow \beta} \) is the corresponding number that ended in state \( \beta \). If no completed segment has been generated in the past, we assume that segments will remain in \( \alpha \), i.e., \( p_{\alpha \rightarrow \beta} = \delta_{\alpha \beta} \). Under this approach, the virtual-end strategy, the next segment will be scheduled in a given state according to the (sampled) probability that the database runs out of segments in that state by the time all presently pending segments are completed. In other words, we are using a meta-simulation to try to predict where the trajectory will end when the segment we are about to schedule is completed and ready to splice. In other words, the actual-end strategy corresponds to scheduling additional segments in the state where the spliced trajectory ends at that particular point in time, while the virtual-end strategy corresponds to scheduling additional segments in a state where the spliced trajectory is likely to be by the time the segments in question are ready to be spliced. The determination of which states are likely is done through an auxiliary simulation where the end states of incomplete segments are randomly sampled based on the statistics gathered during the simulation. Of course, any additional information on the
possible pathways can be leveraged to make that prediction more accurate. Note that the fidelity of this auxiliary simulation does not affect the accuracy of the ParSplice trajectory, but only the computational efficiency.

This strategy is ideally suited to address the low-barrier problem. Indeed, if a trajectory is stuck in a super-basin, it will revisit states within the super-basin many times. Hence, in a statistical sense, its future end point becomes easier to predict. In this case, given enough processors, ParSplice will simultaneously generate segments in all of these states in the exact proportion in which they are expected to be needed. If, on the other hand, the trajectory is not trapped in a super-basin but instead continuously visits new states, ParSplice will naturally revert to the conservative choice of generating all segments at the current end of the trajectory. The algorithm does not make risky bets: speculation only occurs when there is a good statistical reason to do so.

**Computational considerations**

Crucial to the efficiency and simplicity of ParSplice is the ability to label states so that segments can quickly be stored and retrieved in a database without having to sequentially attempt to match the current configuration to every other known one. To achieve that, we follow and make use of labeling of local connectivity graphs. In a nutshell, we generate a connectivity graph where nodes correspond to atoms and edges connect nodes whose corresponding atoms are neighbors in real space, i.e., whose distance is smaller than some preset cutoff chosen to encompass nearest neighbors. The graph is then hashed into an integer value that acts as a state label. The graph can also be topologically sorted before hashing, in which case we obtain a so-called canonical label that is the same for every state that is equivalent up to renumbering of atoms.

An important optimization stems from the fact that the end of a spliceable segment is by construction a valid starting point for another segment. Therefore, each producer stores a copy of the configuration at the end of each segment it generates. If a subsequent request comes in for one of these states, the configuration can be retrieved, thereby saving the cost of having to run for (at least) $t_c$ before the beginning of the segment (the red section in Fig. 1) in order to generate a spliceable starting point.

**Relationship to ParRep**

As mentioned above, ParSplice can be seen as a generalization of the ParRep method (see Ref. [17] for a recent review). In ParRep, the trajectory begins with the system evolving in some state. This trajectory is integrated forward in time until it stays within the same state for a time $t_c$. This is the *decoration* stage. At this point, the configuration of the system is broadcasted to $N - 1$ other replicas. Each one of these new replicas then proceeds to run dynamics until the system remains within the initial state for at least $t_c$. If a trajectory escapes to a different state before $t_c$, it is replaced in the initial state and dephasing is reattempted. This is the *dephasing* stage. After its own dephasing is complete, each replica then enters the *parallel* stage. This stage continues until the first replica (in wall-clock time) observes a transition to a new state. At this point, the cycle repeats and the replica that found a transition enters the decoration
stage. At each cycle, the simulation clock is incremented by the sum of the decorrelation time and of all the parallel time accumulated by the replicas; time is therefore accrued in parallel at a rate that can reach up to $N$ times that of MD. By the same arguments we discussed above, this very simple procedure can nonetheless be made arbitrarily accurate\textsuperscript{6}.

While the upper bound of the parallel speedup of ParRep and ParSplice over standard MD is given by the number of replicas $N$, actual performance can often be much lower. Indeed, as dephasing does not contribute to the simulation time, the parallel efficiency of ParRep is limited by the relative duration of the dephasing ($\sim \tau_c$) and parallel ($\sim \tau_{esc}/N$) stages, with $\tau_{esc}$ a typical escape time, as each replica needs to invest at least $\tau_c$ in overhead to dephase in each state while it can expect to run for $\sim \tau_{esc}/N$ before a transition occurs. Therefore, the number of processors that can be efficiently leveraged (and hence the potential parallel speedup) is roughly given by $\tau_{esc}/\tau_c$. When states are deep, ParRep can leverage massive computational resources: simulations reaching into the ms have been carried out on petascale computers using tens or even hundreds of thousands of cores\textsuperscript{17,19,20}. However, the efficiency is low when $N$ is large and/or when $\tau_{esc}$ is small (relative to $N\tau_c$). It is hence clear why a super-basin composed of shallow states is a problem: because each state is shallow, ParRep will deliver only a modest acceleration over MD, no matter the number of processors used; therefore, it might be impossible to reach the much longer timescales required to escape from a super-basin. Note that the definition of shallow depends on the number of replicas, so that efficiency is increasingly in peril at extreme computational scales where $N$ can be in the tens of thousands or more. In contrast, ParSplice can quickly distribute segments across all states in the super-basin, hence being limited by the much longer super-basin escape time $\tau_{esc}^{sb}$ instead of $\tau_{esc}$. In other words, the ideal ParSplice efficiency limit corresponds to each replica investing $\sim \tau_c$ in overhead to create a spliceable beginning to a segment, but would have to do so only once for each visit to a super-basin, whose duration will typically be much greater than $\tau_{esc}$. While this ideal limit might be hard to approach in practice for shallow super-basins --- as the information required to optimally schedule segments will typically not be known a priori, but needs to be gathered as the simulation proceeds --- significant performance gains over ParRep can be expected, as will be shown below. It is important to note that this performance gain is realized without having to identify super-basins, or even to detect that the simulation might be trapped in such a super-basin.

4. Demonstration and Validation
In order to validate and demonstrate the power of ParSplice, we first consider a prototypical superbasin system: a trimer of silver on a silver (100) surface. The system contains 259 atoms. The different states of the system are defined as the basins of attraction of distinct minima on the potential energy surface, i.e., all points in configuration space that relax to the same energy minimum through a local energy minimization are in the same state. As illustrated by the arrows in Fig. 2, trimers can "spin" through adatom jumping to symmetry-equivalent locations. Such a trimer can spin a large number of times before an adatom detaches from the trimer by jumping to an inequivalent location. This move can lead to a "hop" of the trimer, which will then promptly start spinning again in its new location. Since the energy barrier for spinning is much lower than that for detachment and hopping, the 12 different states that can be connected by adatom jumping (see Fig. 2) form a super-basin. (Note that if atom numbering is ignored, then there are just four different states.) The lower the temperature, the more times a trimer spins before it hops. Temperature thus provides an adjustable parameter to tune the strength of the superbasins. For example, at 650K, each of the 12 states in the super-basin is only visited about 4 times before the trimer hops, while at 500K, each state is visited about 50 times. Additional revisits can occur if the trimer hops back to a previously visited position, but the chain of such revisits can be broken by an adatom exchanging position with a surface atom.
Figure 3: State-occupation autocorrelation function for an unbroken trimer state at T=650K. The relative error vs MD is shown by the continuous lines (right axis).

We validate the method by considering the evolution of the system at 650K. Dynamical accuracy is quantified here by a state-occupation auto-correlation function \( s(t) \), which is defined as the probability that the system is found in a given reference state at time \( t+\tau \) given that it was located in the same reference state at time \( \tau \). The results corresponding to an unbroken trimer state is reported in Fig. 3. The results show a quick decay that correspond to transitions out of the reference state. The subsequent plateau corresponds to stays within the trimer super-basin (where the trajectory has a significant probability of jumping back to the reference state), while the final decay corresponds to escapes from the super-basin through a trimer breakup and hop.

In the reference calculation, \( s(t) \) is computed by direct MD. In ParSplice, we use \( \tau_c = 1 \) and 2 ps and segments are returned as soon as they are at least 1 ps long and are spliceable. As shown in Fig. 3, the agreement is extremely good. Despite the weakly metastable nature of the dynamics at this elevated temperature, ParSplice captures the details to a very high accuracy, both at short (single-transition) times, and long (multi-transitions) times. The relative errors are at the level of 4% for \( \tau_c = 1 \) ps and 1% for \( \tau_c = 2 \) ps.

In the range of temperatures we investigate here (500 to 650 K), the average transition time remains short (between 16 and 49 ps) because the transition barrier for spinning is very low. This means that traditional AMD methods would struggle, as the separation of timescales between \( \tau_c \) (~1ps) and \( \tau_{esc} \) is modest. This is illustrated in Fig. 4, where we compare four different approaches to accelerated MD: standard ParRep (which corresponds to the procedure described in the original manuscript \(^3\) and in the previous section), ParRep in which dephasing is avoided if an already dephased configuration is available from a previous visit to that state,
ParSplice with scheduling at the actual end of the trajectory, and ParSplice with scheduling at the virtual end. The computational speedup is here defined as the simulation time achieved by ParSplice or ParRep divided by that of a standard MD simulation of the same wall-clock duration. In all cases, $\tau_c = 2$ ps and transition detection occurs every 1 ps. In ParRep, the transition time (and computational speedup) is corrected by assuming that it (statistically) occurred at a random time since the last transition check.

The results show a qualitatively different behavior for ParSplice than for ParRep. E.g., at 650K (c.f. left panel), the speedup of conventional ParRep saturates at about 7 (roughly $\tau_{esc}/\tau_c$), because the computational speedup is limited by the number of processors that can efficiently be put to use to escape from a single shallow state. Recycling dephased points improves the performance by roughly a factor two, but still saturates around 15 by the time 128 replicas are used. In contrast, ParSplice with the “actual end” scheduling strategy can deliver speedups in excess of 60 (saturation is not achieved by 128 producers). One also notes that the benefits of the virtual end scheduling strategy increases with the numbers of producers; it delivers a speedup of 75 at 128 producers. This is caused by the fact that the “actual end” scheduling strategy tends to locally generate an excess of segments in states in which splicing is momentarily not possible; these cannot all be consumed unless this state is revisited a sufficient number of times. As the number of revisits is only four here, some segments are left unused at the end of the simulation. Note however that these could potentially be used in a subsequent simulation on the same system as a computational battery, as it was demonstrated in a different context. While the virtual-end strategy is advantageous here, it is not perfect as it requires a
few transitions out of each state to infer a decent model of the distribution of possible segment end states. Using a putatively perfect scheduling strategy, it should be possible to reduce the overhead cost to about $N\tau_c$ for each super-basin, yielding a speedup of about 100 for 128 producers in this case. While the current strategy falls slightly short of this limit, because it needs to learn about the states and their connectivity before making valid predictions, its simplicity is extremely appealing.

The results at 500K confirm the ability of ParSplice to efficiently exploit super-basins. While the speedup accessible with ParRep now saturates at 25 (45 if dephasing is skipped for revisited states), ParSplice is still going strong at 256 producers, producing a speedup of about 230. Note that both scheduling strategies perform equally well here, as expected of deeper super-basins. This clearly shows that the performance of ParSplice is much less sensitive than ParRep to the depth of individual states as long as they form a super-basin. Instead, its performance is limited by the number of revisits to each state. Note that at 650K, the super-basin itself is shallow, as only four revisits per state are expected before a hop. Even in that case, ParSplice significantly outperforms ParRep. However, in the limit where each state is visited only once during a simulation, ParSplice would basically reduce to ParRep in terms of performance, but with additional algorithmic advantages that enhance scalability and fault tolerance. The great strength of ParSplice is that it naturally benefits from the presence of super-basins, without even having to identify or characterize them explicitly.

5. Application

![Figure 5: Illustration of structures of the Ag$_{42}$Cu$_{13}$ cluster. Blue: copper atoms; Red: silver atoms. Left: Mackay icosahedron; Right: 6-fold D6d core.](image)

We demonstrate ParSplice by simulating the evolution of a small Ag/Cu cluster. These so-called nanoalloys are extremely versatile and tunable, as they exhibit a very wide range of chemical and structural order. Such particles are of great interest for a range of applications, from catalysis to medicine\textsuperscript{21}. In the case of Ag/Cu, the two component metals are weakly miscible and so tend to separate, commonly forming core-shell structures with copper at the core. The energy landscape of such nanoparticles, as well as that of other intermetallics, has been
extensively studied theoretically, mainly using global optimization methods (see Ref. [21] for a recent review). These studies are aimed at identifying the most stable structures by searching for low energy basins. They have identified so-called “magic” clusters that are minima of the formation energy at fixed size, w.r.t. composition. For example, it appears that so-called Mackay and anti-Mackay icosahedra are particularly stable, partly due to the lattice mismatch between Ag and Cu. Studies of the dynamics of such clusters are comparatively less common, but the behavior of similar clusters has been investigated by MD \(^{22-23}\). These studies conclude that kinetic trapping in metastable states can come into play for growth on short timescales; it is however unclear in general how strong this effect can be on experimental timescales. To address that question, we here concentrate on the dynamics of a cluster at fixed size and composition, namely \(\text{Ag}_{42}\text{Cu}_{13}\), for which the putative ground state is the Mackay icosahedron (c.f. Fig. 5). Simulations are initialized from a configuration obtained by first melting the cluster and then quenching it rapidly. As in the previous section, states are defined as basins of attraction of the potential energy surface. The correlation time \(\tau_c\) is set to 1ps and segments are at least 10ps in duration; 180 replicas are used. Unless mentioned otherwise, the virtual-end scheduling strategy is used. Because of memory limitations, up to 4000 configurations corresponding to spliceable ends (and consequently beginnings) of segments are stored on each replica. States that are identical upon renumbering of atoms are considered to be equivalent splicing points. An EAM potential is used to describe the interatomic interactions \(^{24}\).

![Figure 6: Distribution of the duration of visits to individual states (left panel) and of the number of visits to individual states (right panel) at T-500K.](image)

Simulations were carried out at temperatures of 475, 500, and 550K and the results are summarized in Table 1. These high temperatures, taken together with the roughness of the landscape, make this system very challenging to simulate with traditional accelerated MD methods. For example, at 500K, the trajectory contains 131408675 transitions over 334 \(\mu s\), for an average transition rate of 2.54 transitions/ps. As shown in Fig. 6, the typical visit is very short; about 90% of the visit last less than 10 ps, and more than half last less than 1 ps. Escape out of these states is therefore not a rare event, and hence cannot be accurately accelerated with conventional AMD (or KMC) methods. However, in ParSplice, these fast transitions are precisely accounted for, since they can only occur, by construction, in the interior of segments. Adding to the difficulty is the extremely large number of states visited by the trajectory: again at
500K, the trajectory visited about 120,000 different states, even after accounting for symmetries. Further, results indicate that we did not fully cover the thermally accessible states, as the number of visited states is still climbing at the end of the simulation.

Despite these challenging conditions, ParSplice performs very well on this system. The efficiency of the simulation in terms of the fraction of the MD effort that contributed to the spliced trajectory ranges from 77.3% at 550K to 93.8% at 475K. Between 16% and 2.6% of the MD effort is unused and still stored in the database as part of spliceable segments at the end of the runs. Less than 7% of the effort is irremediably wasted generating spliceable starting points for segments. The efficiency of the overall simulation as compared to a straight MD simulation is lowered somewhat further due to additional overhead, e.g., for transition checks. We estimate this to account for less than about 5% of the simulation time. Using ParSplice, we were able to simulate the evolution of the system over hundreds of microseconds with a greater-than 150-fold speedup out of a possible 180. Note the significantly lower efficiency of the “current end” (CE) scheduling strategy: it drops to 65.5%, mainly due to a large increase in unused MD time. This is caused by the fact that the CE strategy concentrates the effort in a few states at the time, therefore generating an excess of segments when these states are unlikely to be visited often enough in the future to amortize the investment.

<table>
<thead>
<tr>
<th>run</th>
<th># of blocks generated</th>
<th>% used</th>
<th>% unused</th>
<th>% wasted</th>
<th>trajectory time (μs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>T=475K</td>
<td>820336763</td>
<td>93.8</td>
<td>2.6</td>
<td>3.6</td>
<td>769</td>
</tr>
<tr>
<td>T=500K</td>
<td>375511117</td>
<td>88.9</td>
<td>6.6</td>
<td>4.5</td>
<td>334</td>
</tr>
<tr>
<td>T=500K CE</td>
<td>320369727</td>
<td>65.5</td>
<td>26.7</td>
<td>7.8</td>
<td>210</td>
</tr>
<tr>
<td>T=550K</td>
<td>684746519</td>
<td>77.3</td>
<td>16.0</td>
<td>6.7</td>
<td>529</td>
</tr>
</tbody>
</table>

Table 1: Performance metrics for ParSplice simulations of Ag_{42}Cu_{13}. One block correspond to 1ps of MD time. Used blocks were spliced into the trajectory, unused blocks are part of valid segments that are left in the database at the end of the run, and wasted blocks are blocks that were spent generating valid starting points for segments, and so are not part of any spliceable segment.

For this system, ParSplice is able to exploit the fact that, despite the very large number of states visited and the very short average visit time, some states are revisited extremely often and have long aggregate residence times (c.f. Fig. 6). In fact, 99% of the simulation time is spent in around 2000 states. These act as anchors to and from which generated segments are very likely to be spliced.

The analysis of the trajectories shows that the lowest energy state of the >100,000 that were visited is indeed the Mackay icosahedron with an Ih point group, shown in Fig. 5. However, at 500K, the system spends even more time in other states, most notably states where the core is
of D6d symmetry, composed of two interlocked 6-fold rings with an extra atom in the center, as illustrated in the right panel of Fig 5. 6-fold symmetric cores have been observed for other clusters (e.g., for Ag₃₂Cu₆) but these configurations were not “magic” in the sense defined above. To our knowledge, this is the first time where this configuration is identified to be thermally more stable at elevated temperatures for Ag₄₂Cu₁₃. Indeed, while at the level of individual states, the system spends only marginally longer in any of these 6-fold states than in the ground state; taken together, the former states are dominant, as will be shown below. Each visit to the ground state is however long; the compact, “closed-shell”, structure of the Mackay icosahedron makes the barriers to jump out of that basin quite high. In contrast, visits to the 6-fold states tend to be significantly shorter because the Ag layer around the Cu core of these clusters contains 4 adatom-like “defects” that can diffuse easily. Preliminary simulations we have performed suggest that the ground state of Ag₃₈Cu₁₃ (where these four defects are absent) is of D6d symmetry and that it remains stable at high temperatures.

Figure 7: Harmonic free-energy at T=500K vs potential energy at the basin bottom. The color of the points indicates the symmetry of the Cu core. Green: D6/D6d; Blue: Cs; Red: Ih; Grey: other.

These results point to an entropic stabilization phenomenon, as the energy of the ground state is around 0.6 eV lower than that of these six-fold states. This is confirmed by free-energy calculations in the harmonic approximation, as shown in Fig. 7 for T=500K. The data splits into two bands, the upper one corresponding to icosahedral Ih cores, while the bottom band corresponds to 6-fold symmetric D6/D6d states. The results show that the 6-fold states, despite their higher potential energies, are indeed entropically stabilized by softer phonons, presumably due to their less compact cores. In comparison, the free energy of the ground state is about 0.1
eV above that of the former group. Also note that the states at energies around -138.7 eV correspond to defective icosahedral cores. Again, due to their comparatively hard phonons, these are thermally suppressed by about 0.35 eV at 500K.

Dynamically, the system alternates between two extremely large super-basins that contain the 6-fold and 5-fold icosahedral states. This is illustrated in Fig. 8, where we plot the trajectory at 500K in terms of a hash index that arbitrarily labels each state, as a function of time. Points are colored according to the approximate point group of the Cu core, as determined by the program symmetrizer. Note that the structure of the external shell can subtly break the symmetry of the core. Nonetheless, this analysis proves insightful. The results show that the system alternates between sampling states with mostly 6-fold core (D6 or D6d, green points), and states with cores with mostly Cs symmetry (blue points). The Ih ground state is found amidst the latter (red points). While Cs cores can also be found in 6-fold regions, they tend to concentrate together (blue vertical bands). The gaps in the sequence are a reflection of the long dwell times in some states, in particular in the ground state. The figure clearly shows that the D6 states are dominant and that visits to the other super-basins tend to be short, except when the ground state is encountered. Quantifying this statement precisely is difficult because it would entail classifying defective cores. However, residence in Ih cores accounts for only 7.6% of the time, while unambiguously 6-fold cores (D6, D6d, C6, and C6v) accounts for 45%. Adding states labeled as D2d, whose dominant members were manually confirmed to be 6-fold symmetric, the total exceeds 70%, which is close, but probably still somewhat short of the actual value.

Figure 8: Trajectory at 500K in terms of state indices as a function of time. The color
of the points indicates the symmetry of the Cu core. Green: D6/D6d; Blue: Cs; Red: Ih; Grey: other.

Transitions between the two sets (6-fold vs Cs/Ih) occurs on a timescale of about 5 μs. However, the ground state is not systematically encountered at every time: only 6 independent sets of visits were observed during the simulation, so the 6-fold to Ih transition rate is roughly $2\times10^5 \text{s}^{-1}$, while the reverse transition occurs at a rate of about $8\times10^4 \text{s}^{-1}$. Of course, this relaxation time is expected to grow significantly as the temperature decreases. For example, at 475K, the transitions rates drop to $5\times10^3 \text{s}^{-1}$ and $2\times10^4 \text{s}^{-1}$, respectively. In contrast, transition rates increase to about $1\times10^5 \text{s}^{-1}$ and $2\times10^6 \text{s}^{-1}$, respectively, at 550K. It appears that, at these elevated temperatures, the system achieves thermal equilibrium on sub-millisecond timescales. Assuming an Arrhenius behavior, these rates yield an activation energy of about 0.9 eV for the 6-fold to Ih transition, so that the transition time could reach tens of seconds at room temperature. Note that the initial transient corresponding to the relaxation from the quenched liquid is very short (not even noticeable in Fig. 8), so that the rate limiting step to achieve thermal equilibrium is the transition from 6-fold to Ih structures, not the relaxation from the liquid into either of these two.

6. Discussion

Definition of states and choice of $\tau_c$

A critical ingredient in ParSplice is the ability to define states and estimate a value of $\tau_c$ that will yield a high accuracy without unnecessarily wasting computational resources. The goal is to define states that are strongly metastable, i.e. where the ratio $\lambda_2/\lambda_1$ is as large as possible, so that the probability of escaping the state before $\tau_c$ is low. An obvious choice is to define states as individual basins of attraction of the potential energy landscape, as we have done in the two systems presented here. In this case, estimating $\tau_c$ is greatly simplified by the fact that relaxation occurs on vibrational timescales. However, this simple definition is clearly inadequate for some systems, such as solvated proteins, where the number of individual basins can be astronomical and the vast majority of them are extremely shallow, so that splicing at that level would be hopelessly inefficient. However, ParSplice is formally applicable there, and could become efficient if some more appropriate state partition could be identified. The remaining key question is then to estimate $\tau_c$ for such a partition. This question is still open, but some aspects of the problem are discussed in Ref. [17]. These technical issues at present preclude using ParSplice to simulate systems such as biomolecules without a priori characterizing them in order to define and characterize states, but the formalism that underpins ParSplice is general enough to handle such cases. Note that the concept of “anchor” states, states that are highly recurrent, can be exploited to make ParSplice more amenable to complex systems. If enough of these anchors can be identified, segments can be constrained to begin and end only at these anchors, bypassing the need to partition the rest of configuration space into states. These
anchors however have to be defined in such a way that trajectories have a low probability of running for a long time without encountering one, or the efficiency of ParSplice could suffer.

**Computational aspects**

The algorithmic structure of ParSplice makes it ideally suited for the large-scale, massively-parallel, computing environments which are increasingly common. First, it can be made asynchronous, in the sense that global synchronization of the replicas is not required. This is crucial for scalability at extreme scales. Second, its speculative nature greatly increases the amount of parallelism that is available; this also contributes to potential improvements in scalability. Third, ParSplice offers a simple and efficient approach to deal with faults on producers: as long as producer failures are uncorrelated with the nature of segments (e.g., the probability of failure is independent of whether a transition just occurred), the incomplete segment can simply be purged and the execution continued as if it was never scheduled in the first place. Given these significant advantages, we expect ParSplice to become a very powerful tool that can effectively exploit the next generation of massively parallel computers.

7. **Conclusion**

We have introduced ParSplice, a novel computational approach for the generation of very long atomistic trajectories using massively-parallel computers. A key feature of ParSplice lies in its use of speculation: a “meta-simulation” tries to predict where additional work will be required in the future as the simulation proceeds. Through the introduction of speculation, ParSplice is particularly adept at overcoming the so-called “low-barrier problem” whereby trajectories are trapped in sets of states connected by low-barriers for extended periods of time. Machine learning can be used for accurate speculation of future states, as was recently shown in several systems with heterogeneous barrier distributions. In contrast to other long-timescale methods such as conventional AMD and KMC, ParSplice can remain arbitrarily accurate even while visiting states that are not metastable. In systems where it is possible to define metastable states and estimate their correlation times, ParSplice therefore promises to be an extremely powerful addition to the arsenal of techniques that can be deployed to extend the simulation timescales of complex systems beyond what is possible with conventional MD.

**Acknowledgements**

Work at Los Alamos National Laboratory was supported by the U.S. Department of Energy through the Los Alamos National Laboratory (LANL)/LDRD Program, and by the United States Department of Energy (U.S. DOE), Office of Science, Office of Basic Energy Sciences, Materials Sciences and Engineering Division, during initial conceptual developments. Work at
Harvard University was supported in part by a grant from the U.S. Army Research Laboratory through the Collaborative Research Alliance (CRA) for Multiscale Multidisciplinary Modeling of Electronic Materials (MSME). Los Alamos National Laboratory is operated by Los Alamos National Security, LLC, for the National Nuclear Security administration of the US DOE under contract DE-AC52-06NA25396. This research used resources provided by the Los Alamos National Laboratory Institutional Computing Program.

References


[14] Pande, V.S., Beauchamp, K., Bowman, G.R., “Everything you wanted to know about Markov State Models but were afraid to ask”, *Methods* 2010, 52, 99


