Scalable Cloning on Large-Scale GPU Platforms with Application to Time-Stepped Simulations on Grids

SRIKANTH B. YOGINATH and KALYAN S. PERUMALLA, Oak Ridge National Laboratory

Cloning is a technique to efficiently simulate a tree of multiple what-if scenarios that are unraveled during the course of a base simulation. However, cloned execution is highly challenging to realize on large, distributed memory computing platforms, due to the dynamic nature of the computational load across clones, and due to the complex dependencies spanning the clone tree. We present the conceptual simulation framework, algorithmic foundations, and runtime interface of CloneX, a new system we designed for scalable simulation cloning. It efficiently and dynamically creates whole *logical* copies of a dynamic tree of simulations across a large parallel system without full *physical* duplication of computation and memory. The performance of a prototype implementation executed on up to 1,024 graphical processing units of a supercomputing system has been evaluated with three benchmarks—heat diffusion, forest fire, and disease propagation models—delivering a speed up of over two orders of magnitude compared to replicated runs. The results demonstrate a significantly faster and scalable way to execute many what-if scenario ensembles of large simulations via cloning using the CloneX interface.

Categories and Subject Descriptors: C.2.2 [**Computer Systems Organization**]: Computer-Communication Networks—*Network Protocols*; C.5.1 [**Computer Systems Organization**]: Computer System Implementation—*Large and Medium Computers*; I.6.1 [**Computing Methodologies**]: Simulation and Modeling—*Discrete*; I.6.8 [**Computing Methodologies**]: Simulation and Modeling—*Types of Simulation (Discrete Event, Parallel)*

General Terms: Algorithms, Design, Experimentation, Performance

Additional Key Words and Phrases: Graphical processing units, CUDA, load balancing, time synchronization, what-if decision tree, supercomputing

ACM Reference format:
Srikanth B. Yoginath and Kalyan S. Perumalla. 2018. Scalable Cloning on Large-Scale GPU Platforms with Application to Time-Stepped Simulations on Grids. *ACM Trans. Model. Comput. Simul.* 28, 1, Article 5 (January 2018), 26 pages.
https://doi.org/10.1145/3158669

5

1 INTRODUCTION

1.1 Overview

With many analytic simulation applications, ensembles of simulations are used to explore, test, and experiment with many scenarios. In large-scale scenarios, the simulation in ensembles are often highly related to each other, potentially sharing a large amount of state evolution. Here, we

Authors' addresses: S. B. Yoginath and K. S. Perumalla, Computer Science and Mathematics Division, Oak Ridge National Laboratory, 1 Bethel Valley Rd, Oak Ridge, Tennessee 37831, USA; emails: {yoginathsb, perumallaks}@ornl.gov.
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
2018 Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 1049-3301/2018/01-ART5 $15.00
https://doi.org/10.1145/3158669

focus on ensembles in which simulation scenarios are generated as a dynamically unraveled tree of simulations. Within this simulation tree, each node logically represents an entire simulation of its own, containing its own separate version of simulation state and simulation time starting from its branching point out of its parent simulation. Such ensembles or trees of simulations are characterized by the presence of a large domain size and/or a large number of interacting entities, while the decision points for what-if actions that create branches in the tree involve relatively small, localized, dynamic changes to the scenarios.

Cloning is a conceptual approach in which such a tree of many related simulations is efficiently executed by dynamically minimizing the duplication of memory and computation among the simulations. The efficiency is achieved by separating the logical view and physical manifestations of the simulations in terms of their memory usage and computational operations. Logically, each simulation is an entirely separate simulation of its own. However, using cloning, the *physical* manifestation of the simulations is optimized: the common shared content across state space and virtual time along the *clone tree* hierarchy is combined at runtime, thereby dramatically reducing the aggregate amount of computation and memory consumed by the entire tree.

Although cloning as a concept has been previously proposed and explored (see discussion of related work in Section 1.3), previous work on cloning was restricted to shared memory systems or single-node graphical processing unit (GPU) systems. Problems such as memory management and load balancing were absent in previous implementations that were restricted to a small scale of computing hardware. Modern parallel systems are much larger in size, making it necessary to revisit the cloning techniques to scale to the modern systems.

Toward realizing cloning on modern parallel systems, we present the design, implementation, and performance study of our large-scale, transparent, and optimized simulation cloning framework, which efficiently and dynamically creates whole logical copies of simulations without full physical duplication.

In this article, we focus on advancing the realization of the cloning approach by significantly increasing the scale of cloned simulations. We present the problems and solutions in achieving efficient cloning-based parallel execution of simulations that can scale to the tremendous levels of hardware parallelism offered by supercomputing systems. We describe the conceptual framework, the algorithmic foundations, and a prototype interface with implementation of CloneX as a scalable runtime system for cloning. We also discover and address several performance and scaling challenges that arise in the form of GPU thread management, overall memory management within and outside GPUs, simulation state lookup in the clone tree, and dynamic load balancing of simulation clones at runtime. We also present the performance results from three different benchmark simulation applications—heat diffusion, forest fire, and epidemic outbreak models, which we developed. The performance is studied on up to 1,024 GPUs of a supercomputing system, covering key cloning parameters, namely, branching factors, branching levels, and fraction of common state between clones. Experiments show over two orders of magnitude gains compared to replicated runs, both in terms of memory consumed and computational time.

1.2 Motivation

There are many simulations that do not possess large levels of concurrency in a single run. Yet, it is possible to identify plenty of additional parallelism in the simulation *methodology* and *use-cases* of those simulations.

1.2.1 Computational Gains. The cloning approach addresses multiple parallel computing challenges in the context of utilizing large computing systems as follows.

— Parallelism: As a concurrency amplification technique whose system-level technology is possible to develop in a largely application-oblivious manner, cloning exposes and exploits a new source of parallelism that is otherwise lost in traditional execution flow of simulations. It is ideally suited for a large class of application areas dominated by what-if scenario simulations.
— Memory: Cloning provides tremendous savings in aggregate memory consumed by a non-trivial what-if tree unraveled by dynamically determined decision points.
— Fault Tolerance: Since cloning by definition charts out trajectories that have large fractions of common state, fault tolerant execution is naturally supportable via dynamic redistribution of lost state from live nodes to faulted nodes.
— Real-time Operation: Since cloning permits initiation of multiple branch points with little cost, many solutions may be computed ahead of real-time; the branches that correspond to actual events may be retained when the outcomes of actual events are later available (such a benefit is known to be applicable in real time-constrained simulations such as missile launches).

1.2.2 Applications. The following are intended to illustrate the types of simulation applications in which cloning can be gainfully applied.

— *Forest Fire Simulations*: Simulations of forest fire involve large tracts of land across which fires start and spread, and many alternative scenarios need to be considered, including the possibilities of fires starting at various locations, at various time periods, and in different sequences. Also, effects of many actions need to be modeled, including the intensity of remediation, resource allocations, and schedules of proactive and reactive treatments. Most of these decisions and actions result in localized changes to an otherwise massive simulation. These changes are easily expressed as clones of the base simulation of the entire domain. The clones are incremental changes at various simulation times, all operating on the evolving base scenario.
— *Epidemiological Simulations*: Simulations of the spread of diseases are characterized by large populations forming the base scenario in which the epidemic propagates, while many alternative actions need to be explored, such as curfews, school closures, or vaccinations. What-if analyses of school closures or vaccinations can only be simulated at locations at which high disease intensity can only be found dynamically. Moreover, many hypotheses need to be investigated regarding the locations and intensity of potential sources of new infections, in both space and time. Each proactive campaign or reactive remediation forms a decision point that is localized relative to the base, large-scale domain. Each such decision point can be realized as a clone over the original base simulation.
— *Transportation Simulations*: A large road network forms the base simulation over which many what-if scenarios are explored, such as accident-induced road/lane closures that are dynamically encountered based on congestion levels. While a microscopic vehicular traffic simulation of a city-sized road network forms the base simulation, many local actions or incidents can be explored as incremental changes to the base simulation at various road intersections or times of the day. Cloning is a natural way to define and simulation each such incremental change in a memory-efficient and computationally-efficient fashion.
— *Battlefield Simulations*: In battlefield simulations, all alive/damaged/dead alternatives need to be pursued. For example, branching on the kill probability-based outcomes in battlefield simulations would require full replication of the entire theater of war, while cloning would incur negligible memory cost by simply initiating a few incrementally differentiated clones.

— *Other Simulations*: Similar branching and sharing effects across multiple scenarios arise in other simulations such as climate change (e.g., effects of the choice of clean air or drilling policies in selected countries), material science simulations (e.g., nucleation or fracture initiation), and biochemical simulations (e.g., mutation events or folding events).

1.3 Related Work

The basic concept of cloning in computing may be dated back to John von Neumann's allusion to it for fault tolerance (Von Neumann 1956) in the 1950s; however, not any significant amount of cloning work can be found until the 1990s. During the 1990s, this changed; pioneering work on cloning was performed by Hybinette et al. and was applied successfully to problems such as faster-than-real-time simulation-based decision tools in missile defense applications. The hardware configurations of that time limited the scale to small compute clusters, and the work happened to be primarily focused on parallel discrete event simulations (Bestavros and Wang 1993; Heidelberger 1988; Hybinette and Fujimoto 2001).

While potential benefits of cloning have been known, no large-scale implementation has been realized to date. This is probably (or partly) due to the fact that, at the relatively small scales of parallel computing so far, full replication has not been a major hindrance. Since completely new copies of small simulations could be re-executed for different scenarios, techniques like cloning were not necessarily warranted or sought. This situation has changed lately due to (a) the sizes of individual simulations becoming larger, as in applications simulating millions of entities such as people, vehicles, or geographical regions, and (b) the hardware size of parallel machines becoming larger, as in supercomputers or GPU clusters.

By and large, cloning has generally been restricted to single-node, shared-memory processing, or small scale; consequently, prior work on cloning did not encounter, uncover, or address key challenges that arise on modern parallel platforms with thousands of nodes and many-GPU systems. In our present work, we identify new challenges, such as clone tree management, clone memory management, and fast clone identity lookups as critical concepts that arise in scalable implementations on modern, massively parallel platforms. Additionally, load balancing becomes a key determinant of scalable performance. In this article, we identify and address these issues and challenges in large-scale execution.

Another gap in the literature concerns the theoretical understanding of the computational complexities of cloning. A theoretical analysis of the potential gains of cloning relative to simple ensembles of simulations has been missing in the literature. In this article, we fill this gap by developing a parametrized model of memory and computational gains from cloned execution relative to replicated execution.

Cloning bears superficial similarity with the concepts of transparent duplication for virtual machines in cloud computing, and the copy-on-write semantics of forked UNIX processes, but it is fundamentally different due to the presence of the new dimension of virtual time used in simulation.

Another related approach is that of *computational steering* of the mid/late 1990s (Vetter and Schwan 1997; Vetter and Reed 2000), in which the course of an active simulation may be dynamically altered at runtime; however, such steering is restricted to a single large-scale simulation run and for single, steered trajectory per run.

Cloning has recently been applied to agent-based simulation and its execution on GPUs (Li et al. 2015, 2017). They identified the potential of cloning on GPUs and advanced the state-of-the-art by showing how cloning can effectively help in important applications such as real-time what-if scenarios for crowd management and decision systems. In general, our work is similar to theirs in the fact that both use GPUs and both use cloning. Nevertheless, our work has significant differences

that are complementary or distinguishing. Our CloneX system offers an entirely different set of clone management, memory management, and clone mapping concepts. The concept of load-balancing clones as proposed in this article is novel, which makes it possible to achieve excellent scaling beyond 1,000 GPUs, larger than any scaling of cloned execution previously reported. The parallel implementation has also been able to sustain over 350,000 clones simulated in parallel; to our knowledge, this seems to be one of the largest configurations tested to date. Our implementation and performance study also focuses on memory savings, which is an important gap in the literature that we fill here. In a similar context, new algorithms for distributed execution of cloning have been previously studied (Chen et al. 2003) and applied to federated execution in the context of the High Level Architecture (HLA) (Chen et al. 2005). The latter supports distributed execution; however, it is targeted at interoperable execution, and hence load-balancing considerations and high-performance computing were not specifically considered.

Another recent work that shares some similarity in principle with our present work is the concept of differential simulation (Hanai et al. 2015) (also coauthored by us), which shares similar goals as this article. However, it is fundamentally different in that differential simulation is built on non-volatile memory. In the differential simulation approach, the base simulation is first executed to full completion independently, just like a traditional (single) simulation. However, during the base simulation execution, every event is logged to non-volatile memory, such as a hard disk. This logged trace of events becomes the basis for introducing the desired modifications corresponding to the what-if scenarios. CloneX is a high-performance parallel implementation that is entirely based on computation and inter-processor communication for simulating the complete clone tree and does *not* require disk input/output (I/O).

1.4 Organization

The rest of the article is organized as follows. Section 2 describes the conceptual background and theoretical analysis of cloned execution, along with design considerations for scalable execution of cloning. Section 3 presents the implementation details of the CloneX system for scalable cloned execution on massively parallel GPU platforms. Section 4 provides a detailed performance evaluation study using three benchmarks subjected to cloned execution on up to 1,024 GPUs. The article concludes with Section 5 summarizing the article and identifying future work.

2 CLONING CONCEPTS AND ANALYSIS

2.1 Definitions

Consider any simulation containing models with many interacting entities and a simulation methodology in which *decision points* are encountered in the models. In such simulations, a series of decision points is handled using the traditional approach of replicated runs, which are full-blown physical instances simulations for each trajectory in the series of decisions. However, cloning takes a different approach.

— *Clones*: In cloning, each decision point conceptually results in the generation of two or more different logical instances of the simulation called clones. Although each clone begins a life of its own, it often has very little difference from its parent; large portions of the clone's computation and state remain almost identical to its parent's, the deviation growing only as simulation progresses (or, sometimes, might coincide/collapse entirely into parent's trajectory). For example, in a large-scale traffic simulation with millions of intersections, a clone into which an accident event or lane closure event is introduced continues to share much of the rest of the road network simulation relatively unchanged.

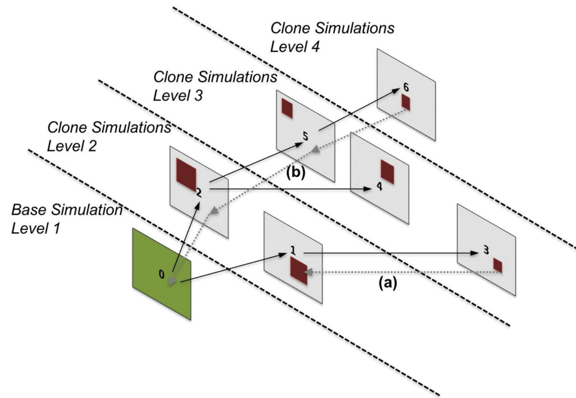


Fig. 1. Illustration of simulation clones and their relationships in the clone tree.

— *Decision Points*: The decision points could be dynamically generated from within the simulation (*endogenous* branches). For example, the ambiguity about kill vs. damage due to a projectile landing in proximity of a battle vehicle creates an endogenous decision point to spawn both trajectories: killed and damaged. Similarly, in epidemiological simulations, the initiation of vaccinations or curfews when disease infection levels cross a threshold are examples of endogenous decision points. Decision points may also be externally introduced (*exogenous* branches). For example, in emergency simulation models, branches may be initiated by user's interest in trying multiple alternatives such as stay-in-place vs. evacuation options. A combination, called *exo-endogenous* decision points, is also possible, for real-time considerations; endogenous branch points are created in anticipation of exogenous input down the line.
— *Clone Tree*: Cloning dynamically creates one or more logical copies of an active simulation even while the simulation is running. These copies start their simulation from the point of the virtual time at which the copies are created. Each logical copy (clone) shares (inherits) all of its logical state from the original simulation state, thereby avoiding having to recompute from initialization to the moment of cloning. The physical differences of the cloned copies (*children*) from the original simulation (*parent*) arise only in the form of decision point values that vary with each copy. A simulation clone can itself become a new original (parent) for another set of clones (for later decision points) at any later time during its own evolution. Thus, in general, cloning creates a tree of simulations, each simulation dynamically charting out its own timeline after branching from its parent, all conceptually sharing state yet logically independent from each other. Also, the simulation clone tree is dynamic in nature as the simulation clones could be created or quashed at various decision points.

Figure 1 is an illustration of the cloned simulation execution framework in action. The *base simulation* is a complete core simulation from which the simulation clones branch off at *decision points*. Logically, the simulation clones are completely different simulations altogether, however physically they correspond to only the $\Delta variations$ (Δ_v) of the *base simulation*. In Figure 1, the small rectangles within the larger ones at every branch represent the memory and computational foot prints of the respective simulation clone, the solid arrow-lines represent the spawning of new simulation clones and the dotted arrow-lines represent the simulation state lookups, which are explained later in Section 2.3.

2.2 Theoretical Analysis

To gain an estimate of the extent to which cloning can help in terms of memory and computation, we developed an abstract, generalized analysis of a tree of cloned simulations. Theoretically, the amount of aggregate memory conserved and the amount of aggregate computation saved due to cloned execution can be roughly estimated for an example cloning scenario. Let this example cloning scenario involve k decision points, m branches, and assume that only the leaf nodes of the clone tree create m simulation clones at every decision point. The base simulation creates m simulation clones at the first decision point and each of these m simulation clones in turn create additional m simulation clones at the second decision point, thus creating a total of $\sum_{l=0}^{k-1} m^l = (m^k - 1)/(m-1)$ simulation clones, where k represents the number of levels and $k = 1$ suggest that only *base- or the root-simulation* clone execution. Hence, for parameter values of $k = 3$ and $m = 3$, a total of 13 simulation clones (including base simulation) will be created. Note, the number of decision points and number-of-levels are the same, since we consider the initiation of base simulation or root simulation clone to based on some user-specific decision, which can also be considered as a initial decision point.

Further, suppose that, on average, only a fraction f of the memory is affected per branch between the parent and child of the decision point, and the computation per unit virtual time of simulation is proportional to the memory affected. Thus, f captures the numerical quantification of Δ_v previously described.

2.2.1 Theoretical Aggregate Memory Savings. The theoretical amount of aggregate memory saved due to cloned execution is roughly estimated[1] here for large values of m and k.

Given a total memory requirement of M bytes per simulation, conventional replicated runs consume Mm^{k-1} bytes of aggregate memory to complete all the runs *concurrently*. The last level k places the most demand for the memory, and all of the previous levels can fit within the maximum requirement of the last level.

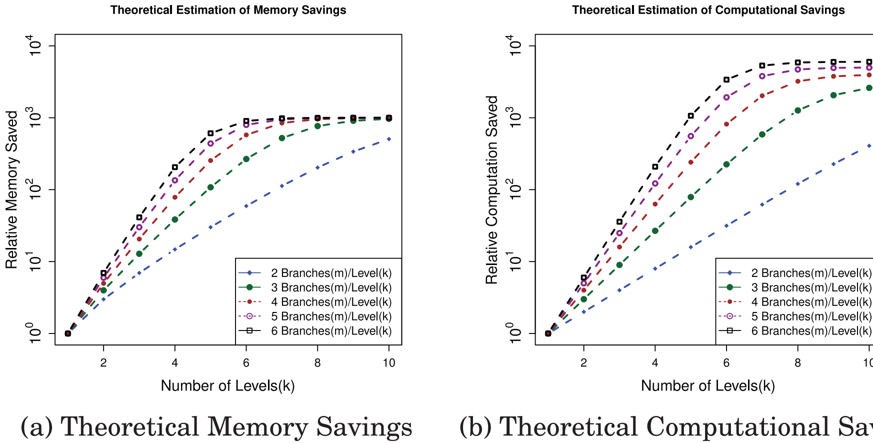
With cloning, only the affected fraction of memory Mf is required per clone, reducing the aggregate memory to $M + Mfm^{k-1}$ $(k > 1)$. Therefore, the expected theoretical factor of reduction in aggregate memory is given by $F_M = 1/(m^{1-k} + f)$. As the number of branches m and decision points k increase, the memory savings becomes large. With even modest values of $m > 2$, and $k \geq 10$, the factor of memory savings F_M tends to equal the inverse of affected memory fraction $f : F_M \approx 1/f$.

In applications where cloning can be used, such as forest fire and epidemiological modeling, simulation scenarios arise by which the cloned decision points involve very small changes, such that $f \leq 10^{-3}$. This indicates that the theoretical reduction in aggregate memory usage compared to conventional replicated simulations can be several orders of magnitude: $F_M \geq 10^3$. For decision points such that $k \gg 10$, or for smaller fractions such that $f \ll 10^{-3}$, the ideal factor of savings is even larger.

Figure 2(a) illustrates the theoretical gains for $f = 10^{-3}$ and $m = 3$, showing the potential to reap orders of magnitude in aggregate memory savings. In the performance study in Section 4, we observe that our implementation in the CloneX system precisely reflects these trends on actual large-scale GPU-based parallel system.

2.2.2 Theoretical Aggregate Computational Savings. Theoretically, the achievable speedup due to cloned executions is roughly estimated as follows. For simplicity, assume that each branching level consumes an equal fraction of the total simulation time, that is, branches from decision points

[1] A more precise formulation that is valid for even small values of m and k is provided in the online supplement.



(a) Theoretical Memory Savings (b) Theoretical Computational Savings

Fig. 2. Illustrative theoretical performance gains from cloning with $f = 10^{-3}$.

occur at equally spaced intervals along the total simulation time. Recollect that, with cloning, all children share the simulation progress (consequently, gain in computation time) of their ancestors before branching, whereas conventional replicated simulations duplicate all computation.

Let C be the total processor time used per simulation instance. After k levels of decision points, each with m branches, the aggregate processor time used for conventional replicated simulations is Cm^{k-1} (for large values of k and m). With cloning, the theoretical aggregate processor time (as k becomes large) is bounded from the top by $C + Cfm^{k-1}/k$. Therefore, the factor of reduction in aggregate computation time is given by $F_C = 1/(m^{1-k} + f/k)$. With conservative estimates, just as in the preceding memory analysis with $f \leq 10^{-3}$, $m > 2$, and $k \geq 10$, the theoretical factor of computational time savings, F_C, is several orders of magnitude. The savings potential becomes even larger for actual large-scale scenarios, just as in the case of savings in aggregate memory usage.

Figure 2(b) illustrates the theoretical computational gains for $f = 10^{-3}$ and $m = 3$, showing the potential to reap orders of magnitude in aggregate computational savings. In the performance study in Section 4, we observe that the cloned execution in our CloneX system implementation precisely reflects these computational savings on actual large-scale GPU-based parallel system.

2.3 Issues and Challenges in Scalable Design

The theoretical analysis of cloned execution indicate the potential for orders of magnitude of gains relative to ensemble runs. To achieve the theoretical potential, the runtime implementation of cloned execution needs to be efficient and scalable. Clone management, synchronization, communication, and computational costs must be minimized to reap the ideal theoretical gains.

Clone Tree Memory Management. Since the simulation clones are only minor variations over their parent simulation clones, they offer the potential to theoretically realize significant memory savings. However, to actually achieve such theoretical savings in practice, we need to limit the memory utilization of the actual simulation clone in software by starting it with only the memory needed to encode the variations from its parent and incrementally grow the clone's simulation state from that of the parent, as needed. We focus on the set of applications in which the simulation state of a modeled entity depends on its immediate neighbors. In these applications, the variations

concentrically grow in layers at every time step. To reflect this growth during cloned simulation executions, the memory of the cloned simulations should be able to expand at every time step. To accomplish this, we need to track *a priori* the extent by which spatial elements in the cloned simulation would grow at every time step. Also, this computation must carefully account for edge effects at the boundaries of the spatial domain, since the newly created simulation clone spatial growth can hit the domain boundaries at any stage of its execution. Additionally, on the GPU platforms, the movement of data from device memory to host memory needs to be minimized to achieve good performance gains.

Clone Tree Dynamic Data Lookup. In parallel execution, each thread of execution evolving a clone needs to lookup and retrieve the simulation state data of the clone's neighbors from its parent clone during execution (data that is logically distinct but not physically instantiated in a clone). With simulation clones spanning several levels of the *clone tree*, the needed information might not be available in its immediate parent clone; this necessitates a lookup up the parental lineage, which, in the worst case, can go up to the root or the base simulation.

A critical point to note is that, even though the root has all the information, a clone cannot always directly pull the data from the root. This is illustrated by the marker (a) in Figure 1: the lookup may be a single hop to its parent simulation clone or may involve multiple hops terminating at the base simulation, as shown by marker (b). To determine the presence or absence of a requested state variable, the lookup mechanism utilizes the span of the clone's Δ_v and the offset (horizontal and vertical) information of the parent clones up the ancestral chain in the clone tree.

Since the lookup operation for each data item is performed extremely often for every time step, the efficiency of the lookup implementation becomes very crucial for achieving overall efficiency from cloning.

Thread Management for Cloned Execution. The runtime performance of modern supercomputing systems employing a large number of GPUs relies on a mode of parallelism called Single Instruction Multiple Data (SIMD) in which many threads apply identical operations on varying streams of data. Modern GPUs provide several thousands of SIMD cores per device and their mapping to the computational problem defines the performance gains. To achieve efficient cloned execution, the GPU threads must be mapped to the varied sizes of cloned domains.

Multi-node Clone Execution and Scaling. Every clone's execution depends on the state of its ancestors. Hence, moving of clones across nodes would also require movement of copies of relevant ancestral clones in the hierarchy up to the root. The parental lineage up to the root needs to be made available at the compute nodes to which the simulation clone is moved. Although this seems burdensome, the operation is not as heavy as it appears, for two reasons. First, the amortized cost of copies of ancestors at any processor will only be logarithmic in the number of local clones. Second, most of the clones (other than the root) only consume a small fraction of the entire simulation space. Hence, making copies of the parents across processors would not be expensive in terms of network communication. In fact, since the root is needed by all clones, a copy of the root needs to be simulated at every node. Hence, to avoid having to transfer the root at runtime, it is better to make a copy of the root at every node during initialization itself. Therefore, in our multi-node execution, the root simulation is hosted on all the execution nodes. The first processor hosts the actual root simulation while other processors also have the same simulation but marked as copies.

To deal with the dynamic growth of the clone tree across multiple computational nodes, we have developed a customized load-balancing algorithm for multi-node simulation clone execution. This load balancer periodically redistributes clones across processors based on the occupied amount of device memory and the number of new clones to spawn at each processor.

3 IMPLEMENTATION

In this section, the data structures and algorithms in the implementation of CloneX are presented. The data is organized across host (main memory) and device (GPU memory) modules, which are synchronized between time steps for the purposes of visualization and instrumentation.

3.1 Overall Execution

The main data structure is an array of “simulation records” analogous to the process table for processes in an operating system. Each simulation record holds the meta information about each clone, such as its identifier, its parent identifier, and its sub-domain specification. The overall execution proceeds as initialization followed by the main simulation loop until the desired virtual time, followed by generation of simulation results and statistics. Within the simulation loop, an optional interactive mode is supported by periodically synchronizing the dynamic simulation state from the device to the host memory and using a graphical interface to display an animation of the simulation. The user interface provides facilities to choose the specific clones to display and navigate across the clone tree.

ALGORITHM 1: Computational loop of CloneX simulation clones

Input:

$t_t, M_1, M_2, S_r, L_t, f_{m1}$

Data:

t_t	Total number of simulation time steps
M_1	List of m1 matrices $[m1_i \ldots]$
M_2	List of m2 matrices $[m2_i \ldots]$
$m1_i$	Matrix1 of simulation clone i
$m2_i$	Matrix2 of simulation clone i
S_r	List of simulation records $[s_i, \ldots]$
s_i	Record $[m1o_i, m2o_i, xo_i, yo_i, nr_i, nc_i, cid_i, pid_i]$
$m1o_i$	Offset to access $m1_i$ in M_1
$m2o_i$	Offset to access $m2_i$ in M_2
xo_i	X offset in simulation clone i
yo_i	Y offset in simulation clone i
nr_i	Number of rows in simulation clone i
nc_i	Number of columns in simulation clone i
cid_i	Clone Id of clone i
pid_i	Parent clone Id of clone i
L_t	Lookup tree
f_{m1}	Flag indicates m_1 is input or output
t_c	Current time step
n_c	Number of clones
cs_i	CUDA stream id

Algorithm:

$t_c = 0$

while $t_c < t_t$ **do**

  Call SpawnClonesIfAny()

  Call LoadBalance()

  for $i \leftarrow 0$ **to** n_c **do**

    Call CloneXKernel

    $(i, cs_i, f_{m1}, m1_i, m2_i, s_i, L_t)$

  end

  $f_{m1} = not f_{m1}$

  if (not *(base simulation))* **then**

    Call AdjustIndices(S_r, f_{m1})

  end

  $t_c = t_c + 1$

end

The simulation records are maintained in a list, represented as S_r in Algorithm 1. Each simulation record contains variables named *m1o*, *m2o*, *xo*, *yo*, *nr*, *nc*, *cid*, and *pid*. The *m1o* and *m2o* hold the offsets into the two matrices (input and output) that the simulation uses at each time step, which will be discussed in more detail in the following memory management section. The *xo* and *yo* are the horizontal and vertical offsets of the two-dimensional simulation space, both of which would be zero for the base simulation, but would be some positive values for any simulation clone that has not grown into its complete spatial dimensions (as large as the base simulation’s full spatial dimensions). The variables *nr* and *nc* represent the spatial dimensions of the simulation clone,

which will always be less than or equal to the spatial dimensions of base simulation. The variables *cid* and *pid* represent the identifier of the clone and its parent, respectively.

The simulation executes for the specified number of time steps t_t and, at each time step, the application's call-back function is invoked to provide the option to the application to spawn any new simulation clones. If the application spawns new clones, then the spawning functionality is carried out by the *SpawnClonesIfAny* of Algorithm 1. Spawning involves two steps: (1) specification of branch conditions and data, (2) distribution of the new clone (simulation records and data) across processors. After the application specifies the data of any new clones, the *LoadBalance* routine is invoked to redistribute the clones for even load balance. This involves identifying lightly loaded processors, assigning the clones to the identified processors, determination of the parents of new clones whose copies should accompany the new clones to the new execution processors, and finally, the transfer of simulation records and the all state values of the new clones and all their associated parents to the destination processors. In this process, care is taken not to needlessly transfer clones whose copies already exist on the destined execution processor.

After the spawn phase, the simulation is advanced by invoking the application model code on every simulation record. Since the clone simulation timelines are independent of one another, we employ CUDA *streams* to parallelize these simulation clone time advancements. The computations are implemented as CUDA kernels and the application makes it available to the simulation clone runtime infrastructure through callback functions. The simulation computations involve reading the previous state information and writing the computed new state information to an output matrix. This is performed by the *CloneXKernel* of Algorithm 1, which is application-specific. After the simulation computations, the f_{m1} flag is set to switch the input and output matrices for next time step computations. We call another CUDA kernel *AdjustIndices* before computing the next time step of the simulation, which updates the simulation records with new *xo*, *yo*, *nr*, *nc*, and *m*1*o* or *m*2*o* values of the new input matrix. This is because clones expand in size at every time step, requiring increase of the memory for the new state, as elaborated next.

3.2 Intra-Node Engine Implementation

Memory Management. The cells on the periphery of the clone simulations need to refer to their parents to obtain the state information of their neighbors. The neighbors' information so read from the parent is added to the simulation clone at every step, which essentially expands the simulation clone spatially.

The spatial expansion requires memory adjustments at every time step. It is highly wasteful to allocate and free memory at every time step to fulfill this requirement. Hence, the maximum amount of available device memory is acquired at the start of the simulation and that is internally managed by CloneX. For computational efficiency, we maintain two copies representing the spatial state information for every simulation clone. At every time step, the previous state of the simulation is read from one copy and the computed state information of the current time step is written to the other. In the next time step, their roles are reversed: the memory copy that was written into previously will be read from and that was read from will be over written with the newly computed state information, as shown in Figure 3.

At every time step, we specify the spatial dimensions of the memory chunk to which state will be written. This requires prior knowledge of the dimension to which the simulation clone expands. Hence, we calculate the memory offsets (*m*1*o* and *m*2*o* in Algorithm 1) *before* the computed state is written to the output matrix. This computation of the offsets must be carefully performed, because the expansion is not always the same in all the directions. To accommodate these considerations, the expansion is calculated in terms of the number of bounding rows and columns that need to be added, which is later converted into the necessary byte information.

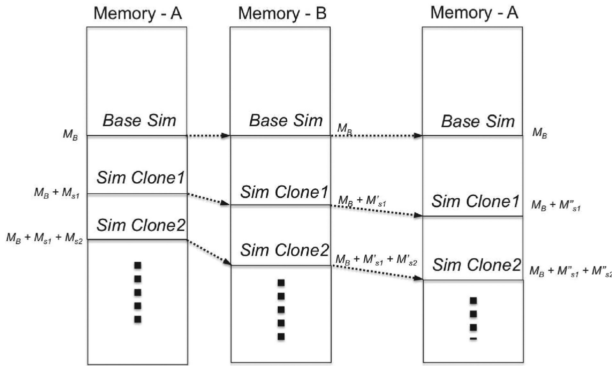


Fig. 3. Simulation clones memory management.

Each simulation clone maintains a simulation record that holds all the necessary information needed to perform its individual simulation computations independently. This record is updated at every time step to reflect the changes in the memory space and the spatial dimensions of the simulation clones. The span of Δ_v of a simulation clone is determined by these $simulationrecord$ variables *xo, yo, nr, nc, cid,* and *pid* (Algorithm 1). During the simulation, only $simulationrecord$ pertaining to the simulation clones are transferred from device memory to the host memory. This information transfer is required because the GPU compute variables like *block* and *grid* dimensions are calculated based on the dimensions of the simulation clone span. The host side of this $simulationrecord$ array is maintained in memory pages that are pinned. The data specific to the simulation-specific state-variables of all the simulation clones stay in device memory and are not transferred at every time step and are only copied during creation or destruction of the simulation clones, or at the end of cloned simulation execution while gathering the results.

GPU Thread Management. To extract maximum utilization of compute resources, we vary the block and grid dimensions to reflect the cloned simulation compute needs. During kernel launching the block dimensions are selected such that it can employ specified number of CUDA threads per block in the computation. The maximum number of threads per block t_{max} is limited by the shared memory that we use. When possible, the entire simulation is computed by a 1×1 grid with block dimensions equal to the simulation stencil dimensions. When the stencil dimensions exceed the number of CUDA threads per block, then the block dimensions are calculated using binary reductions as shown in Algorithm 2.

Algorithm 3 provides the pseudo-code of the *CloneXKernel* of Algorithm 1. This algorithm is executed by each CUDA thread, and hence we contain the number of threads used for computation to the span of the input matrix using the Id_x and Id_y parameters of the CUDA threads. Shared memory of dimension $(bD_x + 2 \times bD_y + 2)$ is populated by values from the input matrix I_m, which could be either m_1 or m_2, based on the flag f_{m1}. While the base simulation populates the shared memory by reading its I_m, the simulation clones that do not span the complete space of the base simulation need to obtain their peripheral values from their parents. The bulk of this algorithm is identifying the threads working on the outer most rows and columns of the input matrix, providing them with additional functionality of getting their neighbors value from parent and writing the values in the shared memory (required for current state update), while also calculating the correct output index to write the computed values to the output matrix. Writing out computed values in the output matrix is performed by the outer SIMD threads using its block dimensions, grid

ALGORITHM 2: Mapping computation onto CUDA threads

Input:
n_r, n_c, t_{max}
Output:
b_x, b_y, g_x, g_y
Data:

n_r	number of rows
n_c	number of columns
t_{max}	CUDA max threads per block
b_x	x dimension of CUDA block
b_y	y dimension of CUDA block
g_x	x dimension of CUDA grid
g_y	y dimension of CUDA grid

Algorithm:
```
if ((n_r × n_c) ≤ t_max) then
    b_x = n_c; b_y = n_r; g_x = 1; g_y = 1
else
    i = 0; j = 0
    tn_r = n_r + (n_r%2)
    tn_c = n_c + (n_c%2)
    while tn_c ≥ √t_max do
        tn_c = tn_c/2 + (tn_c/2)%2
        i = i + 1
    end
    while tn_r ≥ √t_max do
        tn_r = tn_r/2 + (tn_r/2)%2
        j = j + 1
    end
    b_x = tn_c b_y = tn_r g_x = 2^i g_y = 2^j
end
```

ALGORITHM 3: CloneXKernel pseudo-code

Input:
$M_1, M_2, S_r, L_t, f_{m1}$
Data:

Id_x, Id_y	Global thread id along X and Y
I_m, O_m	Input and output matrices
sh_m	Shared mem$[(bD_x + 2) \times (bD_y + 2)]$
bD_x	Block dimension along X
bD_y	Block dimension along Y

Functions:

$GU(v, sh_m)$	Get v from parent and populate sh_m
$DW(v, O_m)$	Compute out index and write v to O_m

Algorithm:
```
m_1 = M_1[S_r.m1_o]; m_2 = M_2[S_r.m1_o]
if f_m1 then
    I_m = m_1; O_m = m_2
else
    I_m = m_2; O_m = m_1
end
if (Id_x < n_c & Id_y < n_r) then
    Populate sh_m with values from I_m
    if not (basesimulation) then
        if within boundary then
            GU(v, sh_m)
            DW(v, O_m)
        end
    end
    Call Simulation_Step(this_cell_val, sh_m)
    Compute out_index
    Write this_cell to O_m
end
```

dimensions, $m1o$ or $m2o$, nr, nc, and additional SIMD thread centric offset values calculating exact position of the output value. The functions DW and GU perform these functionalities.

First, the memory block comprising the state values of the concerned simulation clone is placed inside the shared memory. The block within the shared memory comprises of all the elements whose state-values need to be updated along with the elements that form the circumference of this block. The elements that form the circumference of the block are read-only and are used for computing the state-values of the other elements in the shared-memory. In the case of base-simulation, the values of the elements in circumference are readily available. However, for the cloned-simulations that expand at every simulation time-step, the state-values of the circumference

elements will not be part of cloned simulations and hence, need to be fetched from their parents in the clone-tree hierarchy. This fetching of state-values is performed by *GU* function, which uses the GPU-based Adaptive Radix Tree (GART) for looking up the ID of the closest parent simulation-clone that holds the state-values of these circumferential elements and updates the shared-memory with the state-value read from the parents. Following which, the new state-values of the all the block elements other than circumferential elements are updated. This state-values of a block of cloned-simulation space in the shared-memory need to be written to the global-memory. Since, the cloned-simulation expanded from its previous dimension, the new memory-indices for each of its elements in the block of cloned simulation space needs to be determined. The function *DW* performs this task, where each CUDA-thread computes its output-index and the updates of cloned simulation state-space happen element-by-element and block-by-block, ensuring correctness during expansion of the state-space.

In addition, we also need to accommodate the concern that the expansion of the clone is dependent on its current position in the base simulation span. Based on its position, the expansion could vary from one to all four directions in the two-dimensional simulation space, leading to unequal dimensions of Δ_v span of the simulation clone. Memory expansion and update happens after every time step. The expansion size is determined and the corresponding simulation record of every clone is updated. These updated simulation record values are used during calculation and updating of simulation states during next time step.

Dynamic Lookup. To provide fast lookup operations, we developed an efficient radix-tree-based lookup mechanism that maintains the clone tree on the GPU. Since clones are dynamically distributed across processors, the clone identifiers and their spans do not necessarily offer any pattern to exploit in the clone-to-memory mapping. We implemented an efficient GPU-based index search system (Alam et al. 2016) that returns the memory index given the clone identifier. The GPU-based Adaptive Radix Tree (GART) system has been benchmarked to deliver a very high speed of over 600 million lookups, and returning each index lookup in a few nanoseconds. The main instance of the adaptive radix tree is stored in CPU. All the insert and update operations to add or modify the tree are performed in CPU. GPU is used mainly for the lookup operations. Typically, the number of insert/update operations on the tree is negligible compared to the lookup operations. To efficiently perform the lookup operation on the GPU, the whole tree is copied to the GPU memory. However, CPU-based adaptive radix trees use dynamic memory allocation and use pointers to connect the tree nodes. The next level of the tree could be found only by following pointers. This structure does not work well on the GPU. To get the best performance, memory access should be coalesced. Therefore, the tree is serialized as a byte array for the GPU lookup. The index tree used for searching is only updated when simulations clones are added or removed. The index tree is directly queried from the *CloneXKernel* CUDA kernel to retrieve the information from simulation clone's parental hierarchy.

3.3 Inter-Node Engine Implementation

As cloned simulations progress on the GPUs, the limitation of total available device memory can start constraining the number of simultaneously executing clone simulations. To overcome this memory constraint of a single GPU, multiple GPU nodes will have to be utilized. Thus, our approach is critically motivated by the goals of scaling the number of clones supported and the amount of speedup achieved in the process. The base (root) simulation is clearly the largest in terms of memory occupancy; moreover, since the movement of simulation clones from one node to another also requires the transfer of their parents, we decided to have the base simulation run on all the execution nodes, because the root is the parent of all parents. This would reduce the

network load significantly, because the base simulation does not need to be transferred; moreover, since all the clones are the children of base simulation, its transfer to new nodes is unavoidable and imminent.

The next step toward the multi-node scaling is in differentiating between the *local* and *non-local* clones. This differentiation is necessary to avoid duplicates of the simulation clones from spawning new clones; also, the presence of duplicate clones (with same *cloneId* and *parentId*) cannot be avoided without significant performance loss in the multi-node executions. The simulation clones are tagged *local* during their spawning phase, and they retain the tags even when they are moved by the load balancer to other nodes. However, when the simulation clones are moved to other execution nodes as parents of newly spawned simulation clones, they are tagged as *non-locals*. Although we start with base simulations on all the GPUs during multi-node execution, the base simulation is local only on the process with rank 0 and is marked as non-local on all the other process ranks.

An important component in the multi-node execution is the load balancer, which decides the movement of simulation clones across different nodes. Algorithm 4 gives the pseudo-code of our load-balancing algorithm. The load balancing starts by one process (the leader) gathering the current device memory occupancy (W_p) and the new clone spawning information (dW_p) from each node. It culminates when the leader makes clone-redistribution decisions, and sends out the *sender* S_p and their *receivers* list R_p to each of the clones. This interface is fixed while the implementation can be varied to experiment with different load-balancing schemes. We explored multiple implementations and converged on an efficient variant. In future, it is possible to easily replace with other algorithms customized for future hardware by changing this single module.

In this current load-balancer implementation variant, the balancing loop iteratively maps the smallest of W_ps (*receivers*) to the largest of the dW_ps (senders). Some number of *receivers* are assigned to a *sender*, where the exact number of *receivers* for a given *sender* is a heuristic, whose

ALGORITHM 4: Load-balancing algorithm

Input:
p, W_p, dW_p
Output:
S_p, R_p
Data:

p	Processor $0 \leq p < P$
P	Total number of processors
W_p	Weight (sum of clone sizes) of p
dW_p	New load (clones) being added
R_p	Set of receivers of load from p
S_p	Sender of load to p

```
for ever do
    for all p do R_p ← { p }; S_p ← p; /*initialize*/
    /*Find the processor with most new load*/
    p_max ← max(dW_p, 0 ≤ p < P)
    /*Find top k minimally loaded processors*/
    P_min ≡ {p_1, . . . , p_k} ← {min(W_{p_i}, 0 ≤ p_i < P)}
    Find p_0 such that W_{p_0} is minimum in P_min
    if W_{p_0} = ∞ OR dW_{p_max} = 0 then
        break; /*No more free processors or work*/
    end
    S_{p_i} ← p_max for all p_i ∈ P_min
    R_{p_max} ← P_min
    for all p_i ∈ P_min do
        /*Take p_i out of further consideration*/
        dW_{p_i} ← 0 /*Prevent becoming sender*/
        W_{p_i} ← ∞ /*Prevent becoming receiver*/

    end
    dW_{p_max} ← 0 /*All offloaded*/
    W_{p_max} ← ∞ /*Prevent becoming receiver*/
end
```

value is empirically determined. In our performance runs, we have fixed it to be 1/4th of number of nodes used in parallel runs.

After receiving the *sender* scalar and a *receiver* vector, each process checks the *sender*, if it is itself then they check on the *receiver* list and start sending its local simulation clones to the *receivers*, after splitting the local clones equally among all the *receivers*. However, if the *sender* is not the same as itself, it blocks in a receive call waiting for the *sender*.

The transfer of simulation clones involves moving the simulation records s_i and the initial values of the simulation clones to the remote host. The parent clones that need to be transferred to the new execution node needs to be determined. To avoid duplication of simulation clones (parent) at the *receiver*, the *sender* sends the list of parent simulation clones to the *receiver*, which negates the simulation clones that are already existing at its end. After receiving the revised list of simulation clones, the sender sends the simulation records and the initial values of non-negated simulation clones. After the parent simulation clones are moved to the *receiver*, the *sender* sends the simulation records and the initial data of the new simulation clones to the *receiver*.

After receiving the simulation records and the initial data of the simulation clones to be spawned, the receiver might add a few of its local simulation clones by populating its S_r simulation record list and the data list of M_1 and M_2 matrices. This completes the transfer of the simulation clones from the *sender*. The *sender*, after sending the simulation clones, might also spawn a few of its own local clones, if assigned to do so by the load balancer.

4 PERFORMANCE EVALUATION

4.1 Hardware and Software

The performance study was conducted on a supercomputing system named Titan hosted by the Oak Ridge Leadership Computing Facility. Titan is a hybrid computing architecture featuring 18,688 compute nodes, a total system memory of 710TB, and Cray's high-performance Gemini network. Each node comprises 16-core AMD Opteron processor with 32GB of host memory and an NVIDIA Tesla K20 GPU accelerator containing 2,688 CUDA cores with 6GB of device memory.

The CloneX software is implemented in C++ for the host processor, CUDA 7.0 for the GPU device, and the Message Passing Interface (MPI) for interprocessor communication and synchronization.

4.2 Performance Benchmark Applications

To evaluate the performance gains in terms of computational speed and memory savings, three different benchmarks are used. The first is a synthetic benchmark to simulate heat diffusion under dynamically induced heat sources, the second being simulation of forest fire propagation, and the third is a complex state machine-based propagation of disease with dynamically imposed decision points such as new outbreak events and mitigation campaigns.

The images produced here for the benchmark simulations are snapshots from the CloneX interactive graphical user interface. In each case, the decision points for branching were provided as exogenous events through the user interface.

4.2.1 Heat Diffusion Simulation. For the first synthetic benchmark, we use a two-dimensional (2D) heat diffusion simulation. This simulates the diffusion of heat according to the model in Equation (1):

$$\rho c \frac{\partial T}{\partial t} = \frac{\partial}{\partial x}\left(k\frac{\partial T}{\partial x}\right) + \frac{\partial}{\partial y}\left(k\frac{\partial T}{\partial y}\right). \tag{1}$$

In Equation (1), ρ is the density of the medium, and c is the specific heat. Forward-time-central-space (FTCS), an explicit finite-difference scheme, is employed for solution, as shown in

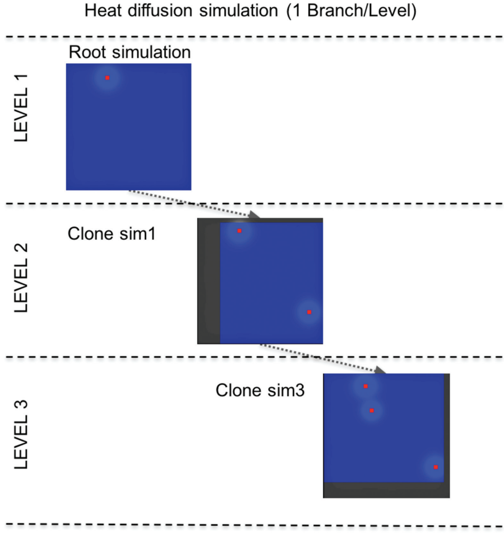


Fig. 4. Visualization of a heat diffusion simulation clone tree.

Equation (2):

$$U_{i,j}^{n+1} = r_x(U_{i-1,j}^n + U_{i+1,j}^n) + r_y(U_{i,j-1}^n + U_{i,j+1}^n) + (1 - 2r_x - 2r_y)U_{i,j}^n. \tag{2}$$

In Equation (22), $r_x = \frac{\sigma \Delta t}{\Delta x^2}$, $r_y = \frac{\sigma \Delta t}{\Delta y^2}$, and $\sigma = \frac{k}{\rho c}$ is the diffusivity of the material, where k is thermal conductivity. Absolute stability of this scheme is assured in the maximum norm when $r_x + r_y \leq \frac{1}{2}$. In particular, when $\Delta x = \Delta y$, the stability requirement is $\sigma \frac{\Delta t}{\Delta x^2} \leq \frac{1}{4}$ (Flaherty 2016). Dirichlet boundary conditions are imposed.

The 2D heat diffusion model is simulated across a thin sheet of iron whose diffusivity $k = 0.23 \times 10^{-6} \mathrm{m}^2\mathrm{s}^{-1}$ across cells of dimension 1cm × 1cm each. Serving as heat sources, a small fraction of the cells in the simulation domain maintain a constant high temperature of 300°C throughout the simulation, while the other cells are initially at a normal temperature of 40°C.

For cloning, each what-if scenario is represented by a simulation clone that is created by randomly picking a part of the domain as a new heat source. Figure 4 illustrates an experimental scenario with one branch per level for three levels in the heat diffusion simulation. It shows a snapshot of simulation clones, where the base simulation is branched off to create simulation clone tagged *Clone sim1* by adding a heat source, which in turn has branched off another simulation adding another heat source to its parent.

4.2.2 Forest Fire Simulation. The second benchmark is a forest fire simulation that follows the model of Balbi et al. (1999), using thermal balance of Equation (3) for simulating the fire dynamics, where $u = T - T_a$, T is the temperature at current simulation time in Centigrade (°C), T_a is the ambient temperature, $\rho_v = \rho_{v0} e^{-\alpha(t - t_{th})}$ (kg/m^2) is the mass of fuel per unit area of platform bed,

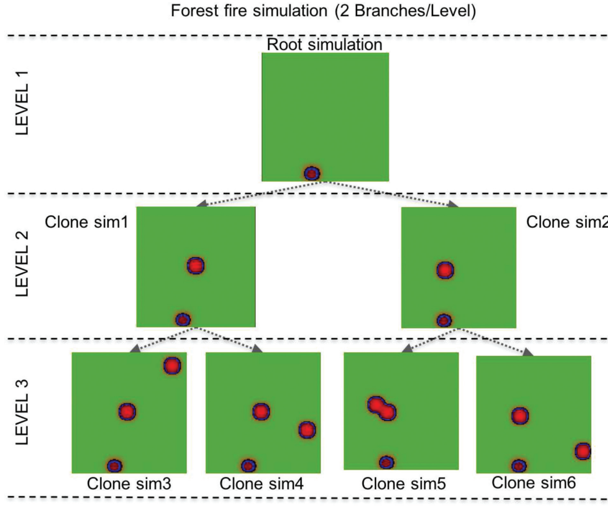


Fig. 5. Visualization of forest fire simulation clone tree.

t is the simulation time in seconds, and t_{th} is the ignition simulation time:

$$\begin{aligned} \frac{\partial T}{\partial t} &= -k(u)) + K\Delta u - Q\frac{\partial \rho_v}{\partial t}, \\ \frac{\partial \rho_v}{\partial t} &= 0, \text{ for unburnt cell,} \\ \frac{\partial \rho_v}{\partial t} &= -\alpha\rho_v, \text{ for a cell reaching ignition temperature.} \end{aligned} \tag{3}$$

As in prior works (Balbi et al. 1999), we set $\rho_{v0} = 0.4\text{kg/m}^2$ and the following values for the constants: $k = 0.071\text{s}^{-1}$, $K = 3.1 \times 10^{-5}\text{m}^2\text{s}^{-1}$, $Q = 3605\text{m}^2\text{Ckg}^{-1}$, and $\alpha = 0.19\text{s}^{-1}$. Further, α (s^{-1}), k (s^{-1}), K (m^2/s) and Q ($\text{m}^2\text{C/kg}$) are constants corresponding to combustion time, convective cooling coefficient, thermal conductivity and enthalpy of combustion, respectively.

The initial conditions are set as $u = 0$ for the unignited cells and boundary cells, and $u = u_{th}$ for the ignited cells, where $u_{th} = T - T_{th}$ and T_{th} is the ignition temperature.

For the size of the simulated domain, the dimension of each cell in the simulation is set to $1\text{m} \times 1\text{m}$. The ignition temperature T_{th} is set to 300°C and other non-ignited cells started at a temperature of 50°C.

The process of cloning involved modeling the ignition of a small block of cells. This is achieved by resetting the selected block of cells to the ignition temperature. Igniting different points in the forest area on fire creates new simulation clones. Other decision point schemes are also possible, such as fire mitigation via watering and fire prevention via dampening.

Figure 5 illustrates cloned simulation executions of two branches/level with three levels in the forest fire simulation scenario. In this figure, we note that the root is simulating the spread of forest fire in a specified geographical area of a domain of user-specified dimension. The simulation clones that are spawned as children of either the base simulation or other clones, at any stage in the simulation, inherit the logical states of all their parents (as described in Section 2). Thus, in this particular forest fire spread simulation scenario, while the fire originating at a certain location is spreading, several what-if scenarios are effectively being evaluated simultaneously. The visual in Figure 5 demonstrates the effect of what-if scenarios corresponding to additional simultaneous fires starting at different locations.

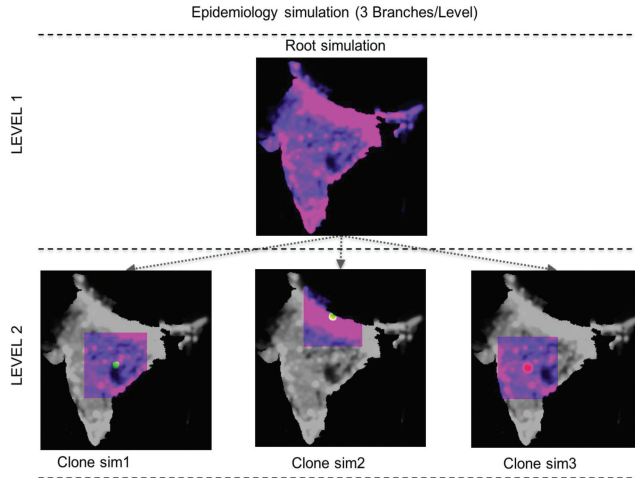


Fig. 6. Visualization of epidemiology simulation clone tree.

4.2.3 Epidemiological Simulation. As the third benchmark, we use an epidemiological model based on geographical population distributions. The geographical domain is divided into cells in which each cell contains four key state variables, each of which is a population count: S for susceptible, E for exposed, I for infected, and R for recovered. This is known as the SEIR model, to which we also add movement of individuals across cells. The initial population densities in the cells are assigned based on population databases of countries made available by the United Nations.

The base simulation tracks the propagation dynamics based on the SEIR model. The clones are spawned based on a variety of what-if scenarios, such as new outbreaks (cells with increased infected count), quarantines (restricted spatial movement), vaccination (reducing susceptible counts), and hospitalization (increasing recovered counts).

Figure 6 illustrates cloned simulations of three branches/level with two levels in the spread and containment of disease in a large geographical area with a high population density. In the benchmarked experimental runs, the population data of India was used for this purpose. The simulation clones spawn increasing numbers of infections and containment zones into the population at runtime to evaluate the what-if disease spread scenarios.

4.3 Performance Parameters

Variables and Constants in Performance Runs. The performance runs are executed for a number of simulation time steps to exercise a sufficient spatial mixture and reach of model dynamics. The key variables in the cloned simulation runs are: the fraction Δ_v of the domain affected that defines each new clone, the number of branches m per level, and the number of levels k. We vary each of these parameters and measure their effects on the runtime performance. A value of $\Delta_v \approx 10^{-3}$ of the input spatial dimension is used in our experiments. The smaller the value of Δ_v, the better the performance of cloning. Hence, the value of $\Delta_v \approx 10^{-3}$ may be considered as a conservative one for bechmarking, because cloned scenarios with smaller values of Δ_v, such as 10^{-4} to 10^{-6}, are possible to find in the benchmarks as well as other applications. The spatial dimension $W \times H$ with width W and height H of all our simulation experiments is set to 2048×2048 (again, even larger dimensions can be defined for higher resolutions and larger domains, for which cloning would perform even better). For this size and $\Delta_v \approx 10^{-3}$ gives each clone's initial dimensions as

Table 1. Number of Clones Spawned for Given Number of Levels (k) and Branches (m)

	$k=1$	$k=2$	$k=3$	$k=4$	$k=5$
$m=1$	1	2	3	4	5
$m=2$	1	3	7	15	31
$m=3$	1	4	13	40	121
$m=4$	1	5	21	85	341
$m=5$	1	6	31	156	781
$m=6$	1	7	43	259	1,555

$\sqrt{2048 \times 2048 \times 10^{-3}} \approx 64$. Hence, the simulation clones start with 64×64-sized scenario-specific spatial data and if it were to grow both horizontally and vertically, it would reach a size of 264×264 by the end of 100 timesteps.

Clone Tree Specification. The number of total time steps (τ) given is equally divided into specified number of levels (k) and, after every τ/k time steps, m simulation clones are spawned. Each of these simulation clones is unique with regards to the spatial location of incidence of the decision point. This results in spawning and executing $\sum_{i=0}^{k-1} m^i = (m^k - 1)/(m - 1)$ simulation clones. The clones are differentiated as *local* and *non-local*: local clones are owned by the processor while non-local ones are copies of parent clones that are owned by another processor. The aggregate number of *local* clones on all nodes in multi-node execution equals to $(m^k - 1)/(m - 1)$, while the total number of clones *local* and *non-locals* will be greater, depending on the movement of clones at runtime by the load-balancing algorithm. Only local clones can spawn new clones, but non-local clones are replicas that come into being only when the load balancer moves copies of parent clones. Further, in our benchmark runs, only the leaf clones spawn new simulation clones.

Computational and Memory Savings. The computational and memory savings are determined with reference to the resource consumption from fully replicated ensembles traditionally run without cloning. The single simulation runtime (R_S) is that of a base simulation. Similarly, the single simulation memory consumption (M_s) is that of the base simulation. Let R_c and M_c be the runtime and memory consumption, respectively, of cloned executions involving n_c clones. The computational savings are determined as $\frac{R_s \times n_c}{R_c}$ and the memory savings are determined as $\frac{M_s \times n_c}{M_c}$.

4.4 Single Node Performance Results

Computational and Memory Savings. Multiple scenarios are executed with varying number of branches per level m and number of levels k parameters. Table 1 shows the aggregate number of clones generated at each level during cloned execution. The number of levels was evaluated up to $k = 5$, and m was varied from 1 to 6, which together provide good coverage with hundreds of thousands of clones. This experimental setup was used for all simulation benchmarks. Figures 7(a) and 7(b) provide the computational and memory savings results for the heat diffusion benchmark; Figures 7(c) and 7(d) show the same for forest fire, and Figures 7(e) and 7(f) for epidemiology. Also note that the ordinates are drawn in logarithmic scale (log_{10}) for all charts reporting computational and memory savings.

Orders of magnitude improvement are observed in computational savings, exceeding $10^{2.5}$ when the number of clones handled is 1,555, while the memory savings are almost a factor of $10^{2.5}$. Similar trends and savings are seen in all the benchmarks. The computational load and memory

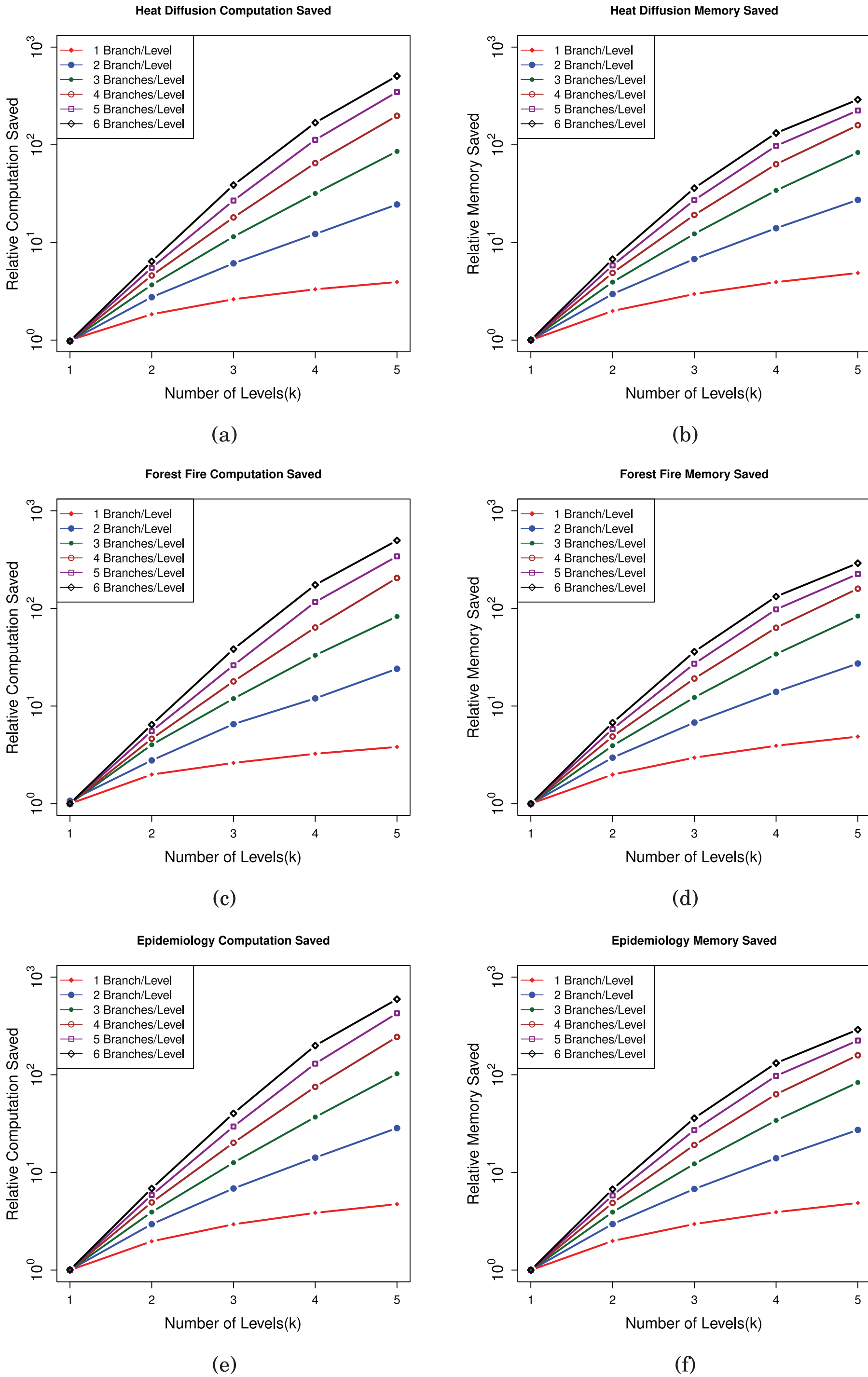


Fig. 7. Factor by which aggregate computation and memory are saved by cloned execution relative to non-cloned execution on a single node ($\Delta_v = 10^{-3}$).

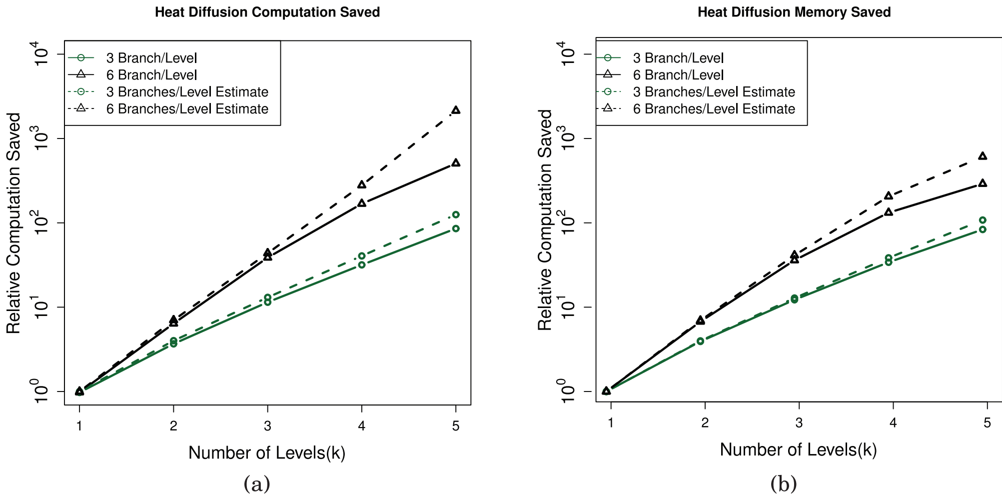
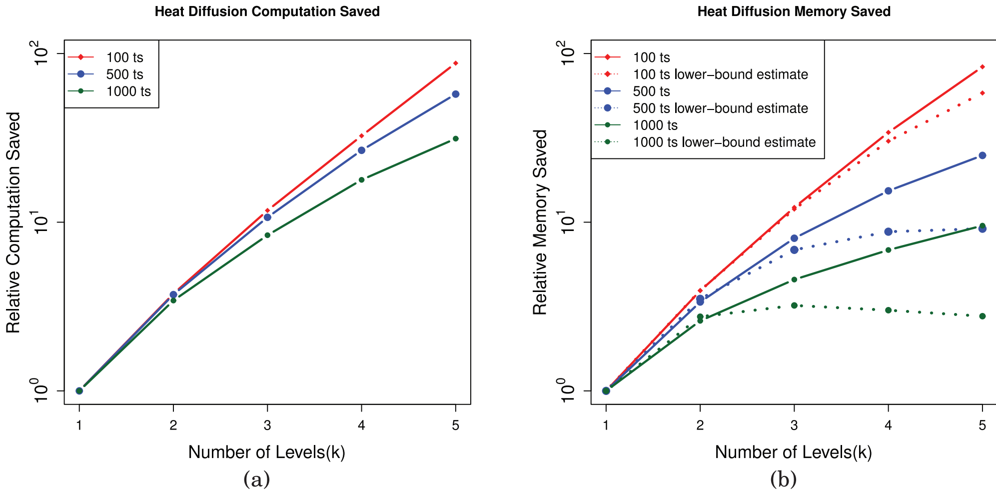


Fig. 8. Theoretically estimated versus actual savings on a single node.

requirements of each of these benchmark simulations are different from one another. However, the spatial dimensions of the base-simulation in each of these benchmarks are same (2048×2048). As the computational and memory trends of cloned simulation runs seen in Figure 7 are relative to their single simulation runs and, the number of clones spawned and span of cloned simulations are the same across different benchmarks, hence similar trends can be expected. Benchmark runs with differing values of m and k can be expected to result in differing trends. These performance trends indicate that the gains due to cloning are not limited to any specific set of simulation applications but apply to wider set of two-dimensional simulation applications that update their states using their immediate neighbor's state information at every timestep.

Comparison with Theoretical Estimation. In Figure 8, against their theoretically expected performance, we plot the computational and memory savings of two scenarios, namely, three Branches/Level and six Branches/Level, with the number of levels varying from 1 to 5. The trends of the observed performance results are nearly identical to theoretical estimates presented in Section 2.2. However, the theoretical expectations assume static clone size and do not consider the growth factor of memory usage as a clone evolves in its timeline after creation. Similarly, the computational increase due the enlarged memory size during simulation is also not considered. Since the total number of time steps is small and the Δ_v is small compared to the full spatial domain of the simulation, this is not evident in Figure 8. In simulations where the number of iterations is comparable to the domain size, the growth effects become important. For this reason, here we derive a more detailed theoretical estimate of the performance that is more precise than the generalized estimate of Section 2.2 that provided an upper bound on the performance under the assumption of a fixed fraction f of clone size relative to the base simulation.

Performance Effects of Timesteps. As expected, in Figure 9, the computational and memory savings decrease with an increase in the number of simulated timesteps, but they are not as drastic as the conservative estimates. Further, compared to memory savings, the computational savings have a relatively milder effect with time steps. This could be attributed to the GPU hardware that



(a) (b)

Fig. 9. Savings (log-scale) in computation and memory on a single node.

provides a low-overhead parallel execution capability; CloneX is seen to be able to harness the capability of exercising concurrent cloned executions.[2]

4.5 Multinode Performance Results

4.5.1 Scaling Behavior. For scaling studies, we ran all of our benchmark simulations for 100 timesteps, with $m = 4$ and $k = 10$, which results in the concurrent execution of 349,525 simulation clones, executed on 128, 256, 512, and 1,024 GPUs.

The summaries of the results from heat diffusion, forest fire, and epidemiology simulations are tabulated in Tables 2, 3, and 4, respectively. In this strong scaling experiment, we observe from the runtime plot in Figure 10 and summary readings in Tables 2, 3, and 4, a good scaling behavior on GPUs up to 1,024. In the tables, μ_{nc} and μ_{nlc} refer to the average number of clones (local + non-local) per GPU and the average number of local clones per GPU, respectively; t_{nc} and t_{nlc} refer to the total number of clones (local + non-local) across *all* the GPUs and the total number of local clones across *all* GPUs, respectively; σ_{nlc} refers to the standard deviation of the number of local clones per GPU; t_s is the runtime in seconds.

4.5.2 Load Balancing. In all the three performance benchmarks runs, doubling the number of GPUs has consistently reduced the runtime by nearly half. The tables also provide an insight on the load distribution across GPUs. The load distribution across the benchmark applications remain similar. A large number of duplicate clones is observed in each simulation benchmark run. Further, the results also show that μ_{nc} and μ_{nlc} halve with doubling of number of GPUs in this strong scaling experiment, while σ_{nlc} remains high.

With multi-node scaling, CloneX simulates nearly 350,000 simulation clones, which is a significantly large number of clones that is impossible to achieve on a single GPU. Also, since the base simulation is large enough to fully utilize all the GPU threads for computation, traditional replicated execution with one simulation clone per GPU of simulations would have required 350,000 GPUs for concurrent processing.

[2]A theoretical estimate of the variation of memory savings with the number of timesteps is provided in the online supplement.

Table 2. Scaling Heat Diffusion Cloning with 4 Branches/Level and 10 Levels

GPUs	μ_{nc}	μ_{nlc}	σ_{nlc}	t_{nc}	t_{nlc}	t_s
128	3,829	2,730	1,306	490,239	349,525	499.46
256	1,958	1,365	696	501,291	349,525	192.53
512	1,021	682	297	523,242	349,525	78.30
1,024	504	341	179	516,101	349,525	27.14

Table 3. Scaling Forest Fire Cloning with 4 Branches/Level and 10 Levels

GPUs	μ_{nc}	μ_{nlc}	σ_{nlc}	t_{nc}	t_{nlc}	t_s
128	3,799	2,730	1,329	486,395	349,525	1,134.30
256	1,948	1,365	693	498,783	349,525	455.33
512	1,022	682	298	523,526	349,525	189.02
1,024	503	341	177	515,885	349,525	65.42

Table 4. Scaling Epidemiology Cloning with 4 Branches/Level and 10 Levels

GPUs	μ_{nc}	μ_{nlc}	σ_{nlc}	t_{nc}	t_{nlc}	t_s
128	3,808	2,730	1,297	487,459	349,525	1,117.25
256	1,942	1,365	685	497,345	349,525	407.79
512	1,029	682	284	526,917	349,525	227.52
1,024	499	341	175	511,723	349,525	65.85

Figure 11 provides the load distribution in terms of (a) the number of clones hosted by each GPU, and (b) the amount of memory occupied used by each GPU, for multi-node heat diffusion execution. The trend labeled "Diffusion 1024" corresponds to cloned execution of Heat Diffusion on 1024 GPUs, and "Diffusion 512" corresponds to that on 512 GPUs, and so on. The load distribution trends in forest fire and epidemiology simulations are similar to those for heat diffusion, and hence omitted here. From Figure 11, we see that the loads become more and more well balanced as the number of GPUs increases, up to a large number of GPUs.

Based on the observation that the load distribution improves with increase in the number of GPUs, the load-balancing algorithm seems to be well suited for large-scale runs.

The limit on the number of receiving GPUs for each GPU that intends to share its load was set to 8, as that setting provided the best runtime balance. Values smaller than that keep the sender heavily loaded, while values larger than that introduce a large overhead in communication and non-local clone copies. Thus, an increase or decrease in the number of receivers in the load-balancing algorithm yields interesting dynamics and insights that need to be further explored. A detailed performance study of our load-balancing algorithm is planned as future work.

5 SUMMARY AND FUTURE WORK

A theoretical analysis has been presented to identify the potential for orders of magnitude of performance gains obtainable by cloned execution of simulations. The design principles have been discussed to achieve the theoretically projected performance. The methodology to efficiently execute cloned simulations across multiple nodes were presented. Algorithms have been designed here to make cloning runtime scalable on a large number of computational nodes, using a load-balancing

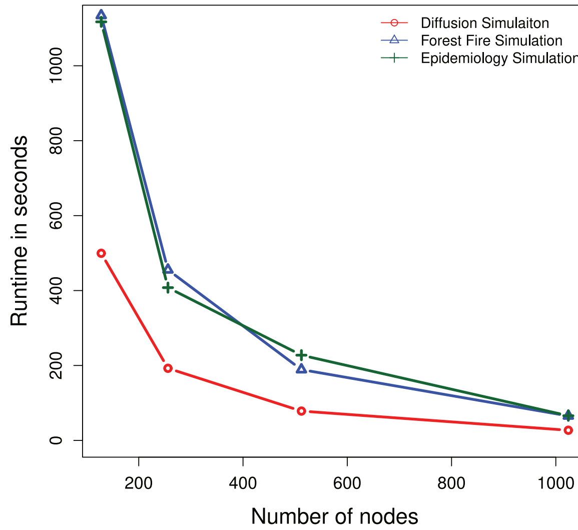


Fig. 10. Cloning scaled to 1,024 nodes (GPUs) simulating 349,525 clones.

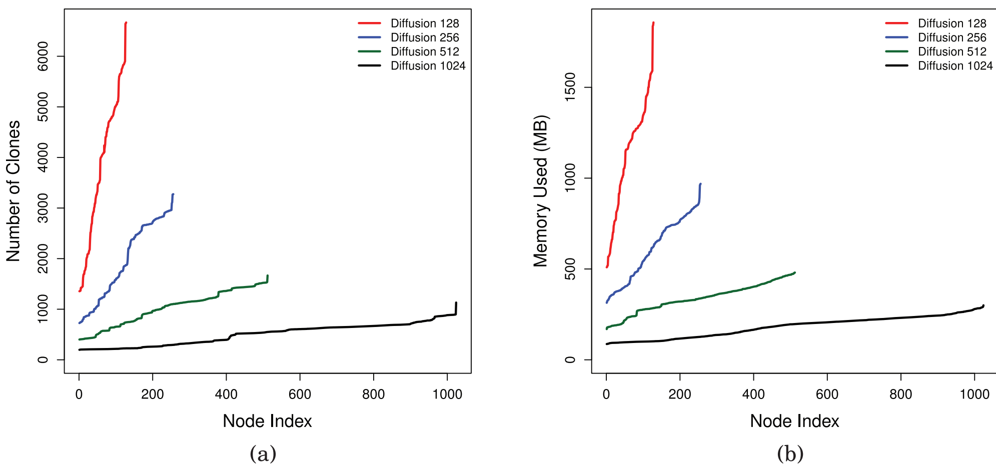


Fig. 11. Load distribution in multi-node parallel execution.

algorithm and clone migration methodology developed specifically for cloned simulation executions. The scalable cloning software data structures and algorithms have been implemented on a supercomputing system. The implementation demonstrated good scaling behavior across multiple GPUs. Gains in excess of two orders of magnitude in computation and memory were observed on three benchmarks: heat diffusion, forest fire, and epidemiological simulations. Future work includes generalizing our approach beyond continuous and time-stepped simulations to parallel discrete event simulations and extending the load-balancing algorithm beyond thousand GPUs. Cloning bears the potential to become a mainstay in all future supercomputing, as a new methodology, a new runtime interface, and a new operating framework in general for many large-scale simulations.

REFERENCES

Maksudul M. Alam, Srikanth B. Yoginath, and Kalyan S. Perumalla. 2016. Performance of point and range queries for in-memory databases using radix trees on GPUs. In *Proceedings of the IEEE Conference on Data Science and Systems.*

J. H. Balbi, P. A. Santoni, and J. L. Dupuy. 1999. Dynamic modelling of fire spread across a fuel bed. *Int. J. Wildl. Fire* 9, 4 (1999), 275–284.

Azer Bestavros and Biao Wang. 1993. *Multi-version Speculative Concurrency Control with Delayed Commit.* Technical Report. Boston University Computer Science Department.

D. Chen, S. J. Turner, W. Cai, B. P. Gan, and M. Y. H Low. 2005. Algorithms for HLA-based distributed simulation cloning. *ACM Trans. Model. Comput. Simul.* 15, 4 (2005), 316–345.

D. Chen, S. J. Turner, B. P. Gan, W. Cai, J. Wei, and N. Julka. 2003. Alternative solutions for distributed simulation cloning. *Simulation* 79, 5–6 (2003), 299–315.

Joseph E. Flaherty. Accessed on: 2016. Multi-Dimensional Parabolic Problems. Retrieved from http://www.cs.rpi.edu/~flaherje/pdf/pde5.pdf.

Masatoshi Hanai, Toyotaro Suzumura, Georgios Theodoropoulos, and Kalyan S. Perumalla. 2015. Exact-differential large-scale traffic simulation. In *Proceedings of the ACM SIGSIM Conference on Principles of Advanced Discrete Simulation.*

Philip Heidelberger. 1988. Discrete event simulations and parallel processing: Statistical properties. *SIAM J. Sci. Stat. Comput.* 9, 6 (1988), 1114–1132.

Maria Hybinette and Richard M. Fujimoto. 2001. Cloning parallel simulations. *ACM Trans. Model. Comput. Simul.* 11, 4 (Oct. 2001), 378–407.

Xiaosong Li, Wentong Cai, and Stephen John Turner. 2015. Cloning agent-based simulation. In *Proceedings of the ACM SIGSIM Conference on Principles of Advanced Discrete Simulation.* ACM, 173–182.

Xiaosong Li, Wentong Cai, and Stephen John Turner. 2017. Cloning agent-based simulation on GPU. *ACM Trans. Model. Comput. Simul.* 27, 2, Article 15 (2017), 15:1–15:24 pages.

Jeffrey Vetter and Karsten Schwan. 1997. High-performance computational steering of physical simulations. In *Proceedings of the 11th International Parallel Processing Symposium.* IEEE, 128–132.

Jeffrey S. Vetter and Daniel A. Reed. 2000. Real-time performance monitoring, adaptive control, and interactive steering of computational grids. *Intl. J. High Perf. Comp. Appl.* 14, 4 (2000), 357–366.

John Von Neumann. 1956. Probabilistic logics and the synthesis of reliable organisms from unreliable components. *Auto. Studies* 34 (1956), 43–98.

Received January 2017; revised August 2017; accepted November 2017