Contents lists available at ScienceDirect

Computer Physics Communications

journal homepage: www.elsevier.com/locate/cpc

Causal set generator and action computer[☆]

William J. Cunningham [a,*], Dmitri Krioukov [b]

[a] Department of Physics, Northeastern University, 360 Huntington Ave. Boston, MA 02115, United States
[b] Departments of Physics, Mathematics, and Electrical & Computer Engineering, Northeastern University, 360 Huntington Ave. Boston, MA 02115, United States

ARTICLE INFO

Article history:
Received 26 September 2017
Received in revised form 29 January 2018
Accepted 4 June 2018
Available online 21 June 2018

Keywords:
Causal sets
Lorentzian geometry
CUDA
x86 Assembly

ABSTRACT

The causal set approach to quantum gravity has gained traction over the past three decades, but numerical experiments involving causal sets have been limited to relatively small scales. The software suite presented here provides a new framework for the generation and study of causal sets. Its efficiency surpasses previous implementations by several orders of magnitude. We highlight several important features of the code, including the compact data structures, the $O(N^2)$ causal set generation process, and several implementations of the $O(N^3)$ algorithm to compute the Benincasa–Dowker action of compact regions of spacetime. We show that by tailoring the data structures and algorithms to take advantage of low-level CPU and GPU architecture designs, we are able to increase the efficiency and reduce the amount of required memory significantly. The presented algorithms and their implementations rely on methods that use CUDA, OpenMP, x86 Assembly, SSE/AVX, Pthreads, and MPI. We also analyze the scaling of the algorithms' running times with respect to the problem size and available resources, with suggestions on how to modify the code for future hardware architectures.

Program summary
Program Title: Causal Set Generator and Action Computer
Program Files doi: http://dx.doi.org/10.17632/5k8wjrhgwh.1
Licensing Provisions: MIT
Programming Language: C++/CUDA, x86 Assembly
Nature of Problem: Generate causal sets and compute the Benincasa–Dowker action.
Solution Method: We generate causal sets sprinkled on a Lorentzian manifold by randomly sampling element coordinates using OpenMP and linking elements using CUDA. Causal sets are stored in a minimal binary representation via the `FastBitset` class. We measure the action in parallel using OpenMP, SSE/AVX and x86 Assembly. When multiple computers are available, MPI and POSIX threads are also incorporated.
Additional Comments: The program runs most efficiently with an Intel processor supporting AVX2 and an NVIDIA GPU with compute capability greater than or equal to 3.0.

1. Introduction

There exists a multitude of viable approaches to quantum gravity, among which causal set theory is perhaps the most minimalistic in terms of baseline assumptions. It is based on the hypothesis that spacetime at the Planck scale is composed of discrete "spacetime atoms" related by causality [1]. These "atoms", hereafter called elements, possess a partial order which encodes all information about the causal structure of spacetime, while the number of these elements is proportional to the spacetime volume—"Order + Number = Geometry" [2]. One of the first successes of the theory was the prediction of the order of magnitude of the cosmological constant long before experimental evidence [3], while one of the most recent significant advances was the definition of a statistical partition function for the canonical causal set ensemble Ω [4] based on the Benincasa–Dowker action [5]. This work, which examined the space of 2D orders $\Omega_{2D} \subseteq \Omega$ defined in [6], provided a framework to study phase transitions and measure observables, with paths towards developing a dynamical theory of causal sets from which Einstein's equations could possibly emerge in the continuum limit. Yet the progress along this path is partly blocked on numerical limitations. Since the theory is non-local, the combination of action computation running times, $O(N^3)$,

[☆] This paper and its associated computer program are available via the Computer Physics Communication homepage on ScienceDirect (http://www.sciencedirect.com/science/journal/00104655).
[*] Corresponding author.
E-mail addresses: w.cunningham@northeastern.edu (W.J. Cunningham), dima@northeastern.edu (D. Krioukov).

https://doi.org/10.1016/j.cpc.2018.06.008
0010-4655/© 2018 Elsevier B.V. All rights reserved.

and thermalization times, $O(N^2)$, of Monte Carlo methods used to sample causal sets from the ensemble, result in $O(N^5)$ overall running times, limiting numerical experimentation to causal set sizes N of just tens of elements.

Here we present new fast algorithms to generate causal sets sprinkled onto a Lorentzian manifold and to compute the Benincasa–Dowker action, with an emphasis on how these algorithms are optimized by leveraging the computer's architecture and instruction pipelines. After providing a short background on causal sets and the Benincasa–Dowker action in Sections 1.1 and 1.2, we describe several algorithm implementations to generate causal sets in Section 2. Section 3 presents a highly optimized data structure to represent causal sets that speeds up the computation of the action, Section 4, by orders of magnitude. Section 5 presents an analysis of algorithms' running times as functions of the causal set size and available computational resources. We conclude with a summary in Section 6.

1.1. Causal sets

Causal sets, or locally-finite partially ordered sets, are the central object in the causal set approach to quantum gravity [1,7,8]. These structures are modeled as directed acyclic graphs (DAGs) with N labeled elements $(n_1, n_2, \dots, n_N)$ and directed pairwise relations (n_i, n_j). If obtained by sprinkling onto a Lorentzian manifold, they approximate the manifold in the continuum limit $N \to \infty$. Lorentzian manifolds are $(d+1)$-dimensional manifolds with d spatial dimensions and one temporal dimension whose metric tensors $g_{\mu\nu}$, $\mu, \nu = 0, 1, \dots, d$, have one negative eigenvalue [9,10]. These DAGs are a particular type of random geometric graph [11]: elements are assigned coordinates in time and d-dimensional space via a Poisson point process with intensity ξ, and they are linked pairwise if they are causally related, i.e., timelike separated in the spacetime with respect to the underlying metric (Fig. 1). As a side note, sprinkling onto a given Lorentzian manifold is definitely not the only way to generate random causal sets. The general definition of a causal set can be found in [1], and random causal sets also can be obtained by sampling from the canonical ensemble Ω [4], or more generally, from the ensemble of random partial orders $P_{n,p}$ [12], i.e., they can in general be treated as unlabeled partial orders. Due to the non-locality implied by the causal structure, causal sets have an information content which scales at least as $O(N^2)$ compared to that in competing theories of discrete spacetime which scales as $O(N)$ [13–15]. As a result, by using the causal structure information contained in these DAG ensembles, one can recover the spacetime dimension [16,17], continuum geodesic distance [18], differential structure [19–22], Ricci curvature [5], and the Einstein–Hilbert action [13,23–25], among other properties.

1.2. The Benincasa–Dowker action

In many areas of physics, the action (S) plays the most fundamental role: using the least action principle [26,27], one can recover the dynamic laws of the theory as the Euler–Lagrange equations that represent the necessary condition for action extremization $\delta S = 0$. In general relativity, from the Einstein–Hilbert (EH) action,

$$S_{EH} = \frac{1}{2}\int R\left(x^\mu\right)\sqrt{-g}\,dx^\mu\,, \tag{1}$$

where R is the Ricci scalar curvature and g is the metric tensor determinant, Einstein's field equations can be explicitly derived and then solved given a particular set of constraints [28]. Therefore, if one hopes to develop a dynamical theory of quantum gravity, one would hope that either the discrete action in the quantum theory converges to (1) in the large-N limit, as we find with the

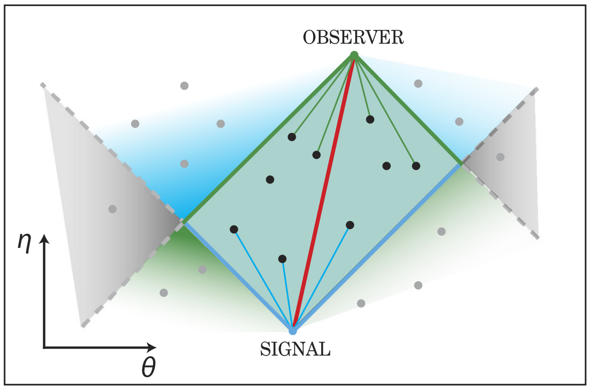

Fig. 1. The causal set as a random geometric graph. Elements of the causal set are sprinkled uniformly at random with intensity ξ into a particular region of spacetime, where η and θ respectively refer to the temporal and spatial coordinates in $(1+1)$ dimensions. Light cones, drawn by 45-degree lines in these conformal coordinates, bound the causal future and past of each element. When light cones of a pair of elements (shown in blue and green) overlap, the elements are said to be causally related, or timelike separated, as indicated by the bold red line. The black elements both to the future of the signal and to the past of the observer form the pair's Alexandroff set shown by the teal color. Not all pairwise relations are drawn. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

Regge action for gravitation [29], or an interacting theory leads to an effective action, as we see with the Wilson action in quantum chromodynamics [30]. The numerical investigation of whether such a transition does indeed take place can be quite difficult: the quantum gravity scale is the Planck scale, so that if the convergence is slow, it may be extremely challenging to observe it numerically. This is indeed the case for the causal set discrete action, known as the Benincasa–Dowker (BD) action [5], which has been shown to converge slowly to the EH action in curved higher-dimensional spacetimes such as $(3+1)$-dimensional de Sitter spacetime [22,24].

The BD action was discovered in the study of the discrete d'Alembertian (B), i.e., the discrete covariant second-derivative approximating $\Box \equiv -\partial_t^2 + \nabla^2$, defined in $(1+1)$ dimensions, for instance, as

$$B\phi\left(x^\mu\right) = \frac{2}{l^2}\left(-\phi\left(x^\mu\right) + 2\left[\sum_{y\in L_1} - 2\sum_{y\in L_2} + \sum_{y\in L_3}\right]\phi\left(y^\mu\right)\right), \tag{2}$$

where $\phi(x^\mu)$ is a scalar field on the causal set, $l \equiv \xi^{-1/(d+1)}$ is the discreteness scale, and the ith order inclusive order interval (IOI) L_i corresponds to the set of elements $\{y\}$ which precede x with exactly $(i-1)$ elements $\{z_j\}$ within each open Alexandroff set, i.e., $y \prec \{z_j\} \prec x \,\forall\, y \in L_i$ and $|\{z_j\}| = i-1$. In [5] it was shown that in the continuum limit, (2) converges in expectation to the continuum d'Alembertian plus another term proportional to the Ricci scalar curvature

$$\lim_{N\to\infty} \mathbb{E}\left[B\phi\left(x^\mu\right)\right] = \Box\phi\left(x^\mu\right) - \frac{1}{2}R\left(x^\mu\right)\phi\left(x^\mu\right)\,. \tag{3}$$

From (2) and (3) one can see when the field is constant everywhere, so that $\Box\phi(x^\mu) = 0$, then (2) converges to the Ricci curvature in the continuum limit, and therefore to the EH action when summed over the entire causal set. It was also shown in [5] that the expression for the BD action in $(1+1)$ dimensions is

$$S_{BD} = 2(N - 2n_1 + 4n_2 - 2n_3)\,, \tag{4}$$

where n_i is the abundance of the ith order IOI, i.e., the cardinality of the set L_i (Fig. 2). While (4) converges in expectation, any typical causal set tends to have a BD action far from the mean. This poses a

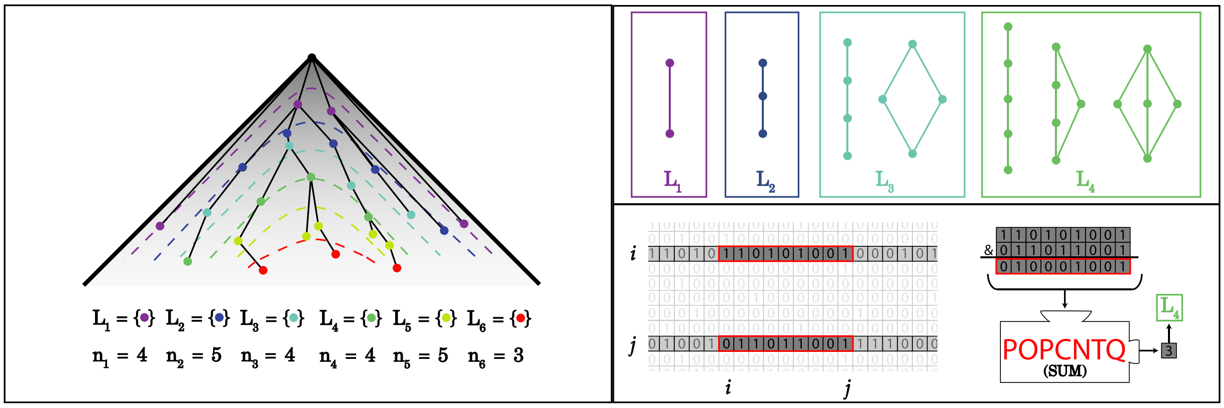
L₁ = {●} L₂ = {●} L₃ = {●} L₄ = {●} L₅ = {●} L₆ = {●}
n₁ = 4 n₂ = 5 n₃ = 4 n₄ = 4 n₅ = 5 n₆ = 3
L₁
L₂
L₃
L₄
i
j
&
POPCNTQ
(SUM)
3

Fig. 2. Proper distance and the order intervals. The left panel shows discrete hypersurfaces of constant proper time $\tau = \sqrt{x^2 - t^2}$ (dashed) are approximated using the graph distance. If the black point is some element x in a larger causal set, then the order intervals would be found by counting the number of elements belonging to each hypersurface, i.e., $n_i = |L_i|$. In general the structure is not tree-like. The top of the right panel shows the subgraphs associated with each of the first four inclusive order intervals used in (4), and the bottom part shows how they are detected using the causal (adjacency) matrix, assuming the graph has been topologically sorted, i.e., time-ordered. For each pair of timelike separated elements (i, j), we take the inner product of rows i and j between columns i and j using the bitwise AND in place of multiplication and the `popcntq` instruction in place of a sum. The resulting value tells how many elements lie within the Alexandroff set. Details of the algorithm can be found in Section 3.3.

serious problem for numerical experiments which already require large graphs, $N \gtrsim 2^{16}$, to show convergence, and also indicates that Monte Carlo experiments must have relatively large thermalization times. To partially alleviate this problem, it is not (2) which one usually calculates, but rather another expression, called the "smeared" or "non-local" action (S_ε), which is obtained by averaging (or smearing) over subgraphs described by a mesoscale characterized by $\varepsilon \in (0, 1)$. The new expression which replaces (4) is

$$S_\varepsilon = 2\varepsilon \left[N - 2\varepsilon \sum_{i=1}^{N-1} n_i f_2(i-1, \varepsilon) \right],$$
$$f_2(i, \varepsilon) = (1-\varepsilon)^i \left[1 - \frac{2\varepsilon i}{1-\varepsilon} + \frac{\varepsilon^2 i(i-1)}{2(1-\varepsilon)^2} \right]. \quad (5)$$

The smeared action (5) was shown to also converge to the EH action in expectation, while fluctuations are greatly suppressed so that numerical experiments with the same degree of convergence accuracy can be performed with orders of magnitude smaller graph sizes [22].

While in some cases one might want to compare directly the expectation of the BD action to the continuum result (1), in Monte Carlo experiments with the canonical causal set ensemble one uses (5) in the quantum partition function

$$Z(N, d, \mathcal{T}) = \sum_C e^{iS_\varepsilon/\hbar}, \quad (6)$$

where the sum is over the ensemble of all causal sets C with fixed size N, dimension d, and topology $\mathcal{T}$. The analytically continued partition function used in numerical experiment is

$$Z(N, d, \mathcal{T}) = \sum_C e^{-\beta S_\varepsilon/\hbar}, \quad (7)$$

where $\hbar \to 1$ and $\beta \in \mathbb{R}^+$. Methods for generating causal set Markov chains using this partition function are discussed in [4,31].

1.3. Computational tasks

Generating causal sets involves an $O(N)$ coordinate generation operation followed by an $O(N^2)$ element linking operation, both of which can be parallelized (Section 2). Yet the bottleneck is not graph generation but the $O(N^3)$ action computation. After each causal set is constructed, the primary computationally intensive task in computing (5) is counting the IOIs. For each pair of causally related elements we must count the number of elements within their Alexandroff set. As a result, the runtime depends greatly on the ordering fraction, defined as the fraction of related pairs, which in turn depends on the choice of manifold, dimension, and bounding region.

Previous work implemented as a part of the Cactus Framework [32] has been quite successful, but because the causal set toolkit is part of a broader numerical relativity package it is challenging to modify core data structures and to take advantage of platform-specific architectures. Therefore, one of the main new features of the software suite presented here is a new efficient data structure called the `FastBitset` (Section 3), which offers compressed-bit storage and several highly optimized algorithms designed specially to calculate the smeared BD action. As a result, larger causal sets may be studied in the asymptotic regime $N \gtrsim 2^{16}$, possibly up to the extreme sizes $N \sim 2^{24}$, and the Markov chains generated by smaller causal sets may be extended further than before to enable a closer examination of phase transitions [4,13].

We note that if other possible forms of the causal set action arise in the future, as soon as their definitions rely only on the adjacency matrix of a causal set, they can also take advantage of the presented algorithms, since these algorithms use only causal set adjacency matrices, and rely on optimized set and counting operations. For the same reasons, i.e., since these algorithms use causal set adjacency matrices only, they can be applied without modification not only to causal sets obtained by sprinkling onto a Lorentzian manifold, but also to any other causal sets, e.g., to Kleitman–Rothschild partial orders [33].

2. Causal set generation

2.1. Coordinate generation

For a finite region of a particular Lorentzian manifold, coordinates are sampled via a Poisson point process with intensity ξ, using the normalized distributions given by the volume form of the metric. For instance, for any $(d + 1)$-dimensional Friedmann–Lemaître–Robertson–Walker (FLRW) spacetime [34] with compact spatial hypersurfaces, the volume form may be written

$$dV = a(t)^d dt\, d\Omega_d, \quad (8)$$

where $a(t)$ is the scale factor, which describes how space expands with time, and $d\Omega_d$ is the differential form for the d-dimensional sphere. From this expression, we find the normalized temporal distribution is $\rho(t) = a(t)^d / \int a(t')^d \, dt'$, and spatial coordinates are sampled from the surface of the d-dimensional unit sphere. Because the $(d+1) \times N$ coordinates of the elements sprinkled within a spacetime are all independent with respect to each other, these may easily be generated in parallel using OpenMP, which is a C/C++ and Fortran library used to distribute parallel tasks over multiple CPU cores [35].

2.2. *Pairwise relations*

Once coordinates are assigned to the elements, the pairwise relations are found by identifying pairs of elements which are timelike separated, and efficient storage requires the proper choice of the representative data structure. A causal set is a graph, i.e., a set of N labeled elements along with a set of pairs (i, j) which describe pairwise relations between elements, so the most straightforward representation uses an adjacency matrix of size $N \times N$. If the graph is simply-connected, i.e., there exist no self-loops or multiply-connected pairs, then this matrix contains only 1's and 0's, with each entry indicating the existence or non-existence of a relation between the pair of elements specified by a particular pair of row and column indices. Moreover, if this graph is undirected, the matrix will be symmetric. We represent naturally ordered causal sets as undirected graphs with topologically sorted elements, meaning that elements are labeled such that an element with a larger index will never precede an element with a smaller index. In the context of a conformally flat embedding space, which is the only type we consider in this work, this simply means elements are sorted by their time coordinate before relations are identified. Yet this does not mean that the presented causal set generation algorithms are impossible to adjust to generate causal set sprinkled onto spacetimes that are not conformally flat. Indeed, in such spacetimes topological sorting can be used, as any partial order can be topologically sorted by the order-extension principle [36].

2.2.1. *Naive CPU linking algorithm*

The naive implementation of the linking algorithm using the CPU uses a sparse representation in the compressed sparse row format [37,38]. Because the elements have been sorted, we require twice the memory to store sorted lists of both future-directed and past-directed relations, i.e., one list identifies relations to the future and the other those to the past. While identification of the relations is in fact only $O(N^2)$ in time, the data reformatting (list sorting) pushes it roughly to $O(N^{2.6})$, as we will see in Section 5.

2.2.2. *OpenMP linking algorithm*

The second implementation uses the dense graph representation and is parallelized using OpenMP. Using this dense representation for a sparse graph can waste a relatively large amount of memory compared to the information content; however, the nature of the problem described in the previous section dictates a dense representation will permit a much faster algorithm, as we will discuss later in Sections 4 and 5. Moreover, the sparsity will depend greatly on the input parameters, so in many cases the binary adjacency matrix is the ideal representation.

2.2.3. *Naive GPU linking algorithm*

While OpenMP offers a great speedup over the naive implementation, the linking algorithm is several orders of magnitude faster when instead we use one or more Graphics Processing Units (GPUs) with the CUDA library [39]. Since they have many more cores than CPUs, GPUs are typically best at solving problems which require many thousands of independent low-memory tasks to be performed. There are many difficulties in designing appropriate algorithms to run on a GPU: one must consider size limitations of the global memory, which is the GPU equivalent of the RAM, and the GPU's L1 and L2 memory caches, as well as the most efficient memory access patterns. One particularly common optimization uses the shared memory, which is a reserved portion of up to 48 KB of the GPU's 64 KB L1 cache. This allows a single memory transfer from global memory to the L1 cache so that spatially local memory reads and writes by individual threads afterwards are at least 10x faster. At the same time, an additional layer of synchronizations among threads in the same thread block (i.e., threads which execute concurrently) must be considered to avoid thread divergence [40] and unnecessary `if/else` branching. It also puts constraints on data structures since it requires spatially local data or else the cache miss rate, i.e., the percent of time data is pulled from the RAM instead of the cache, will drastically increase.

The first GPU implementation offers a significant speedup by allowing each of the 2496 cores in the NVIDIA K80m (using a single GK210 processor) to perform a single comparison of two elements. The output is a sparse edge list of 64-bit unsigned integers, so that the lower and upper 32 bits each contain a 32-bit unsigned integer corresponding to a pair of indices of related elements. After the list is fully generated, it is decoded on the GPU using a parallel bitonic sort to construct the past and future sparse edge lists. During this procedure, vectors containing degree data are also constructed by counting the number of writes to the edge list.

2.2.4. *Optimized GPU linking algorithm*

Despite the great increase in efficiency, this method fails if N is too large for the edge list to fit in global GPU memory or if N is not a multiple of 256. The latter failure occurs because the thread block size is set to 128 for architectural reasons,[1] and the factor of two comes from the index mapping used internally which treats the adjacency matrix as four square submatrices of equal size. The second GPU implementation addresses these limitations by tiling the adjacency matrix, i.e., sending smaller submatrices to the GPU serially. Further, when N is not a round number these edge cases are handled by exiting threads with indices outside the proper bounds so that no improper memory accesses are performed.

This second implementation also greatly improves the speed by having each thread work on four pairs of elements instead of just one. Since each of the four pairs has the same first element by construction, the corresponding data for that element may be read into the shared memory, thereby reducing the number of accesses to global memory. Moreover, threads in the same thread block also use shared memory for the second element in each pair. Hence, since each thread block has 128 threads and each thread works on four pairs, there are only 132 reads (128+4) to global memory rather than 512 (128×4), where each read consists of reading $(d+1)$ floats for a $(d+1)$-dimensional causal set. Finally, when the dense graph representation is used, the decoding step may be skipped, which offers a rather substantial speedup when the graph is dense. There are other optimizations to reduce the number of writes to global memory using similar techniques via the shared memory cache.

2.2.5. *Asynchronous GPU linking algorithm*

A third version of the GPU linking algorithm also exists which uses asynchronous CUDA calls to multiple concurrent streams [39]. By further tiling the problem, simultaneously data can be passed to and from the GPU while another stream executes the kernel, i.e., the linking operations. This helps reduce the required bandwidth over the PCIe bus, which connects the GPU to the CPU and

[1] On the NVIDIA K80m, which has a Compute Capability of 3.7, each thread block cannot have greater than 1024 threads, there can be at most 16 thread blocks per multiprocessor, and at the same time no greater than 2048 threads per multiprocessor.

other devices, and can sometimes improve performance when the data transfer time is on par with the kernel execution time. We find in Section 5 this does not provide as great a speedup as we expected, so this is one area for future improvement should this end up being a bottleneck in other applications.

3. The `FastBitset` class

3.1. Problems with existing data structures

The relations found by the linking algorithm are best stored in dense matrix format for the action algorithm, as we will see in Section 4. A binary adjacency matrix can be implemented in several ways in C++. The naive approach is to use a `std::vector<bool>` object. While this is a compact data structure, there is no guarantee memory is contiguously stored internally and, moreover, reading from and writing to individual locations is computationally expensive. Because the data is stored in binary, there is necessarily an internal conversion involving several bitwise and type-casting operations which make these simple operations take longer than they would for other data structures.

The next best option is the `std::bitset<>` object. This is a better option than the `std::vector<bool>` because it has bitwise operators pre-defined for the object as a whole, i.e., to multiply two objects one need not use a `for` loop; rather, operations like `c = a & b` are already implemented. Further, it has a bit-counting operation defined, making it easy to immediately count the number of bits set to '1' in the object. Still, there is no guarantee of contiguous memory storage and, worst of all, the size must be known at compile-time. These two limitations make this data structure impossible to use if we want to specify the size of the causal set at runtime.

Finally, the last option we will examine is the `boost::dynamic_bitset<>` provided in the Boost C++ Libraries [41]. While this is not a part of the ISO C++ Standard, it is a well-maintained and trusted library. Boost is known for offering more efficient implementations of many common data structures and algorithms. The `boost::dynamic_bitset<>` can be dynamically sized, unlike the `std::bitset<>`, the memory is stored contiguously, and it even has pre-defined bitwise and bit-counting operations. Still, it does not suit the needs of the abovementioned problem because it is not possible to access individual portions of the bitset: we are limited to work only with individual bits or the entire bitset.

Given these limitations, we have developed the `FastBitset` class to represent causal sets in a way which is most efficient for non-local algorithms such as the one used to find the BD action. The adjacency matrix is comprised of a `std::vector` of these `FastBitset` objects, with each object corresponding to a row of the matrix. Internally, this data structure holds an array of 64-bit unsigned integers, referred to as blocks, which contain the matrix elements in their raw bits. We have provided all four set operations (intersection, union, disjoint union, and difference) and several bit-counting operations, including variations which may be used on a proper subset of the entire object. The performance-critical algorithms used to calculate the BD action have been optimized using inline assembly and Intel's Streaming SIMD Extensions (SSE) and Advanced Vector Extensions (AVX) instructions [42].

3.2. Optimized algorithms in the `FastBitset`

One of the most frequently used operations in the action calculation is the set intersection, i.e., row multiplication using the bitwise `AND` operator (Fig. 2 (right)). The naive implementation uses a `for` loop, but the optimized algorithm takes advantage of the 256-bit YMM registers located within each physical CPU core [42]. For a review of x86 microarchitectures, see [43,44]. The larger width of these registers means that in a single CPU cycle we may perform a bitwise `AND` on four times the number of bits as in the naive implementation at the expense of moving data to and from these registers. The outline is described in Algorithm 1. It is important to note that for such an operation to be possible, the array of blocks must be 256-bit aligned. Any bits used as padding are always set to zero so they do not affect any results.

Algorithm 1 Set Intersection with AVX

```
Input:
    $A$                ▷ The bit array of the first FastBitset
    $B$                ▷ The bit array of the second FastBitset
    $n$                ▷ The number of blocks
1: procedure INTERSECTION(A,B,n)
2:     for $i = 0;\ i < n;\ i += 4$ do
3:         ymm0 ← $A[i]$
4:         ymm1 ← $B[i]$
5:         ymm0 ← (ymm0) & (ymm1)
6:         $A[i]$ ← ymm0
Output:
    $A$                ▷ The first bit array now holds the result
```

The code shown inside the `for` loop is written entirely in inline assembly, with Operation 5 using the SIMD instruction `vpand` provided by AVX. Therefore, for each set of 256 bits, we use two move operations from the L1 or L2 cache to the YMM registers, one bitwise `AND` operation, and one final move operation of the result back to the general purpose registers. The bottleneck in this operation is not the bitwise operation, but rather the move instructions `vmovdqu`, which limits throughput due to the bus bandwidth to these registers. As a result, it is not faster to use all 16 of the YMM registers, but rather only two. While certain prefetch instructions were tested we found no further speedup.

One of the reasons this data structure was developed was so we could perform such an operation on a subset of two arrays of bits. We apply the same principle as in Algorithm 1, but with unwanted bits masked out, i.e., set to zero after the operation. For blocks which lie outside the range we want to study, they are not even included in the `for` loop. The new operation, denoted the *partial intersection*, is outlined in Algorithm 2.

In the partial intersection algorithm, we consider two scenarios: in one the entire range of bits lies within a single block, and in the second it lies over some range of blocks, in which case the original intersection algorithm may be used on those full blocks. In either case, it is essential all bits outside the range of interest are set to zero, as shown by the `memset` and `get_bitmask` operations.

The final operation which we must optimize to efficiently calculate the action is the bit count and, therefore, the partial bit count as well. This is a well-studied operation which has many implementations and is strongly dependent on the hardware and compiler being used. The bit count operation takes some binary string, usually in the form of an unsigned integer, and returns the number of bits set to one. Because it is such a fundamental operation, some processors support a native assembly instruction called `popcnt` which acts on a 32- or 64-bit unsigned integer. Even on systems which support these instructions, the compiler is not always guaranteed to choose these instructions. For instance, the GNU function `__builtin_popcount` actually uses a lookup table, as does Boost's `do_count` method used in its `dynamic_bitset`. Both are rather fast, but they are not fully optimized, and for this reason we will attempt to package the fastest known implementation with the `FastBitset`. When such an instruction is not supported the code will default to Boost's implementation.

The fastest known implementation of the bit count algorithm uses the native 64-bit CPU instruction `popcntq`, where the trailing 'q' indicates the instruction operates on a (64-bit) quadword

Algorithm 2 Partial Intersection with AVX

```
Input:
    A                                          ▷ The first bit array
    B                                          ▷ The second bit array
    o                                          ▷ Starting bit index
    n                                          ▷ Length of subset
1:  function GET_BITMASK(offset)
2:      return (1 ≪ offset) − 1
3:  procedure PARTIAL_INTERSECTION(A,B,o,n)
4:      ▷ Divide o by 64 to get the block index
5:      $x \leftarrow o/64$
6:      ▷ Indices within the blocks
7:      $a \leftarrow o \% 64$
8:      $b \leftarrow (o + n) \% 64$
9:      if range inside single block then
10:         $A[x] \leftarrow A[x]$ & $B[x]$ & get_bitmask($a$) & get_bitmask($b$)
11:         $u \leftarrow 1$                       ▷ Used one block
12:     else
13:         ▷ Intersection on full blocks
14:         $m \leftarrow (n - 1)/64$              ▷ Number of full blocks
15:         intersection($A[x + 1], B[x + 1], m$)
16:         ▷ Intersection on end blocks
17:         $A[x]$ &= $B[x]$ & get_bitmask($a$)
18:         $A[x + m]$ &= $B[x + m]$ & get_bitmask($b$)
19:         $u \leftarrow m + 2$                   ▷ Used $m + 2$ blocks
20:     ▷ Set other blocks to zero
21:     $l \leftarrow a$
22:     $h \leftarrow$ A.getNumBlocks()$- l - u$
23:     if $l > 0$ then
24:         memset($A, 0, 8 * l$)
25:     if $h > 0$ then
26:         memset($A[l + u], 0, 8 * h$)
Output:
    A                          ▷ The first bit array now holds the result
```

Algorithm 3 Optimized Bit Counting

```
Input:
    A                                          ▷ The bit array
    N                                          ▷ The number of blocks
1:  procedure COUNT_BITS(A,N)
2:      ▷ The counter variables
3:      $c[4] \leftarrow \{0, 0, 0, 0\}$
4:      for $i = 0; i < N; i += 4$ do
5:          $A[i] \leftarrow$ popcntq($A[i]$)
6:          $c[0] += A[i]$
7:          $A[i + 1] \leftarrow$ popcntq($A[i + 1]$)
8:          $c[1] += A[i + 1]$
9:          $A[i + 2] \leftarrow$ popcntq($A[i + 2]$)
10:         $c[2] += A[i + 2]$
11:         $A[i + 3] \leftarrow$ popcntq($A[i + 3]$)
12:         $c[3] += A[i + 3]$
Output:
    $c[0] + c[1] + c[2] + c[3]$                ▷ Number of set bits
```

operand. While we could use a `for` loop with a simple assembly call, we would not be taking advantage of the modern pipeline architecture [44] with just one call to one register. For this reason, we unroll the loop and perform the operation in pseudo-parallel fashion, i.e., in a way in which prefetching and prediction mechanisms will improve the instruction throughput by our explicit suggestions to the out-of-order execution (OoOE) units in the CPU. We demonstrate how this works in Algorithm 3.

This algorithm is so successful because the instructions are not blocked nearly as much here as if they were performed using a single register. This is because the `popcnt` instruction has a latency of three cycles, but a throughput of just one cycle, meaning x `popcnt` instructions can be executed in $x + 2$ cycles instead of $3x$ cycles when they are all independent operations [45]. As a result, the Intel instruction pipeline allows the four sets of operations to be performed nearly simultaneously (i.e., instruction-level parallelism) via the OoOE units. While it would be possible to extend this performance to use another four registers, this would then mean the bitset would need to be 512-bit aligned.

3.3. The vector product

To execute the vector product operation, we want to utilize the features described above. If the `popcnt` is performed directly after the intersection, a lot of time is wasted copying data to and from YMM registers when the sum variable could be stored directly in the YMM registers, for instance. Since the `vmovdqu` operations are comparatively expensive, removing one out of three offers a great speedup. Furthermore, for large bitsets it is in fact faster to use an AVX implementation of the bit count [46]. We show such an implementation below in Algorithm 4.

Algorithm 4 Optimized Vector Product

```
Input:
    A                                          ▷ The first bit array
    B                                          ▷ The second bit array
    N                                          ▷ The number of blocks
1:  procedure VECPROD(A,B,N)
2:      ymm2←table                             ▷ Lookup table
3:      ymm3←0xf                               ▷ Mask variable
4:      for $i = 0; i < N; i++$ do
5:          ymm0← $A[i]$
6:          ymm1← $B[i]$
7:          ymm0←(ymm0) & (ymm1)               ▷ Intersection
8:          ymm4←(ymm0) & (ymm3)               ▷ Lower Mask
9:          ymm5←((ymm0) ≫ 4) & (ymm3)         ▷ High Mask
10:         ymm4←vpshufb(ymm2, ymm4)           ▷ Shuffle
11:         ymm5←vpshufb(ymm3, ymm5)           ▷ Shuffle
12:         ymm5←vpaddb(ymm4, ymm5)            ▷ Horiz. Add
13:         ymm5←vpsadbw(ymm5, ymm7)           ▷ Horiz. Add
14:         ymm6←ymm5+ ymm6                    ▷ Accumulator
15:     $c \leftarrow$ymm6
Output:
    $c[0] + c[1] + c[2] + c[3]$                ▷ Vector product sum
```

This algorithm is among the best known SIMD algorithms for bit accumulation [46]. At the very start, a lookup table and mask variable are each loaded into a YMM register. The table is actually the first half of the Boost lookup table, stored as an `unsigned char` array. These variables are essential for the instructions later to work properly, but their contents are not particularly interesting. Once the intersection is performed, two mask variables are created using the preset mask. The bits in these masks are then shuffled (`vpshufb`) according to the contents of the lookup table in a way which allows the horizontal additions (`vpaddb`, `vpsadbw`) to store the sum of bits in each 64-bit range in the respective range. Finally, the accumulator saves these values in `ymm6`. The instructions are once again paired in a way which allows the instruction throughput to be maximized via instruction-level parallelism, and the partial vector product uses a very similar setup to the partial intersection with respect to masking and `memset` operations. If the bitset is too short, i.e., if the causal set is too small, this algorithm will perform poorly due to the larger number of instructions, though it is easy to experimentally determine which to use on a particular system and then hard-code a threshold.

All of the algorithms mentioned so far may be easily optimized for a system with (512-bit) ZMM registers, and we should expect the greatest speedup for the set operations. Using Intel Skylake X-series and newer processors, which support 512-bit SIMD instructions, we may replace something like vpand with the 512-bit equivalent vpandd. An optimal configuration today would use a Xeon E3 processor with a Kaby Lake microarchitecture, which can have up to a 3.9 GHz base clock speed, together with a Xeon Phi Knights Landing co-processor, where AVX-512 instructions may be used together with OpenMP to broadcast data over 72 physical (288 logical) cores.

Algorithm 5 Optimized Cardinality Measurement

```
Input:
    A                                   ▷ The adjacency matrix
    c                                   ▷ The array of cardinalities
    p                                   ▷ The number of element pairs
1: procedure CARDINALITY(A,c,p)
2:     for $k = 0$; $k < p$; $k$ ++ do
3:         ▷ Convert the pair index to two element indices
4:         $\{i, j\}$ ←convert_index($k$)
5:         if elements are not related then
6:             continue
7:         ▷ Cardinality for pair $(i, j)$
8:         $m$ ← $A[i]$.partial_vecprod($A[j]$, $i$, $j - i + 1$)
9:         $c[m + 1]$ ++
Output:
    c                                   ▷ The populated array
```

4. Action computation

4.1. Naive action algorithm

The optimizations described above which use AVX and OpenMP are orders of magnitude faster than the naive action algorithm, which we review here. The primary goal in the action algorithm is to identify the abundance n_i of the subgraphs L_i identified in Fig. 2. When we use the smeared action rather than the local action, this series of subgraphs continues all the way up to those defined by the set of elements L_{N-2}, i.e., the largest possible subgraph is an open Alexandroff set containing $N - 2$ elements. Therefore, the naive implementation of this algorithm is an $O(N^3)$ procedure which uses three nested for loops to count the number of elements in the Alexandroff set of every pair of related elements. For each non-zero entry (i, j) of the causal matrix, with $i < j$ due to time-ordering, we calculate the number of elements k both to the future of element i and to the past of element j and then add one to the array of interval abundances at index k.

4.2. OpenMP action algorithm

The most obvious optimization uses OpenMP to parallelize the two outer loops of the naive action algorithm, since the properties of each Alexandroff set in the causal set are mutually independent. Therefore, we combine the two outer loops into a single loop of size $N(N - 1)/2$ which is parallelized with OpenMP, and then keep the final inner loop serialized. When we do this, we must make sure we avoid write conflicts to the interval abundance array: if two or more threads try to modify the same spot in the array, some attempts may fail. To avoid this, we generate T copies of this array so that each of the T threads can write to its own array. After the action algorithm has finished, we perform a reduction on the T arrays to add all results to the first array in the master thread. This algorithm still scales like $O(N^3)$ since the outer loop is still $O(N^2)$ in size.

4.3. AVX action algorithm

The partial vector product algorithms described in Section 3.3 naturally provide a highly efficient modification to the naive action algorithm. The partial intersection returns a binary string where indices with 1's indicate elements both to the future of element i and to the past of element j, and then a bit count will return the total number of elements within this interval. A summary of this procedure is given in Algorithm 5.

This algorithm is able to be further optimized by using OpenMP with a reduction clause (which prevents write conflicts) to accumulate the cardinalities. In turn, each physical core is parallelizing instructions via AVX, and then each CPU is parallelizing instructions by distributing tasks in this outer loop to each core. While it is typical to use the number of logical cores during OpenMP parallelization, we instead use the number of physical cores (typically half the logical cores, or a quarter in a Xeon Phi co-processor) because it is not always efficient to use hyperthreading alongside AVX.

4.4. MPI optimization: static design

When the graph is small, so that the entire adjacency matrix fits in memory on each computer, we can simply split the for loop in Algorithm 5 evenly among all the cores on all computers using a hybrid OpenMP and Platform MPI approach. But when the graph is extremely large, e.g., $N \gtrsim 2^{21}$, we cannot necessarily fit the entire adjacency matrix in memory. To address this limitation, we use MPI to split the entire problem among 2^x computers, where $x \in \mathbb{Z}^+$. Each computer will generate some fraction of the element coordinates, and after sharing them among all other computers, will generate its portion of the adjacency matrix, hereafter referred to as the adjacency submatrix. In general, these steps are fast compared to the action algorithm.

The MPI version of the action algorithm is performed in several steps. It begins by performing every pairwise operation possible on each adjacency submatrix, without any memory swaps among computers. Afterward, each adjacency submatrix is labeled by two numbers: the first refers to the first half of rows of the adjacency submatrix on that computer while the second corresponds to the second half, so that there are 2^{x+1} groups of rows labeled $\{0, \ldots, 2^{x+1} - 1\}$. There will never be an odd number since the matrix is 256-bit aligned. We then wish to perform the minimal number of swaps of these row groups necessary to operate on every pair of rows of the original matrix. Within each row group all pairwise operations have already been performed, so moving forward only operations among rows of different groups are performed.

We label all possible permutations except those which provide trivial swaps, i.e., moves which would swap the submatrix rows in memory buffers within a single computer, or moves which swap buffers in only some computers. The non-trivial configurations are shown for four computers in Table 1. By organizing the data in this way, we can ensure no computer will be idle after each data transfer. We use a cycle sort to determine the order of permutations so that we can use the minimal number of total buffer swaps. We are able to simulate this using a simple array of integers populated by a given permutation, after which the actual operation takes place. By starting at the current permutation and sorting to each

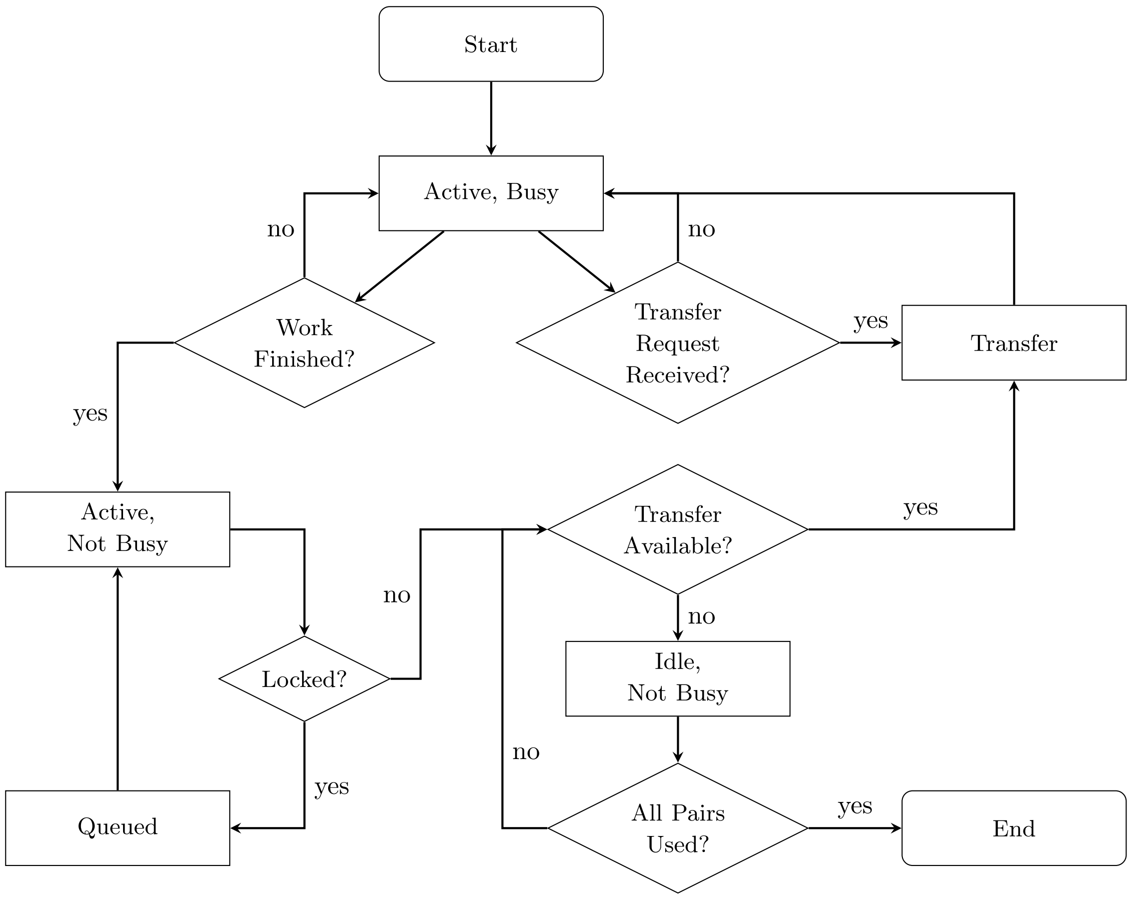

Fig. 3. Load-balanced action algorithm using MPI. When the adjacency matrix is split among multiple computers, we want to make sure no computers end up idle for long periods of time, yet to move from an Idle to Busy state at least one other computer must have finished its work. Initially, all computers are Active and Busy, indicating they are not waiting for another task to finish and are currently working on the action algorithm. If two other computers have requested an exchange, an Active, Busy computer will allow them to use part of its memory for temporary storage (Transfer). Once a computer finishes its portion of work on the action algorithm, it will enter the Active, Not Busy state, at which point it will add its pair of buffer indices to the global list of available buffers. An MPI spinlock, developed specifically for this algorithm, is implemented to ensure only one computer can manage a transfer. If another pair of computers is exchanging data, the Active, Not Busy computer will enter a Queued state, where it will remain until other transfers have completed. Otherwise, it will attempt a memory transfer if possible by checking the list of available buffers. If no other buffers are available, or if any available transfers would lead to redundant calculations, the computer enters the Idle, Not Busy state, where it waits for another computer to initiate a transfer. Once all buffer pairs have been used, the algorithm ends.

Table 1
Permutations of MPI buffers using four computers.

Rank 0		Rank 1		Rank 2		Rank 3	
0	1	2	3	4	5	6	7
0	3	2	5	4	7	6	1
0	5	2	7	4	1	6	3
0	7	2	1	4	3	6	5
0	2	1	3	4	6	5	7
0	4	1	5	2	6	3	7
0	6	1	7	4	2	5	3

Each of four computers, identified by its rank, holds a quarter of the adjacency matrix. Two buffers on each computer each hold an eighth of the entire matrix, labeled $\{0, \ldots, 7\}$, so that all pairwise row operations may be performed using the minimal number of inter-rank transfers. Each of the seven rows is a non-trivial permutation of the eight buffers, indicating only six rounds of MPI data transfers are necessary to calculate the action when the algorithm is split over four computers.

unvisited permutation, we can record how many steps each would take. Often it is the case that several will use the same number of steps, in which case we may move from the current permutation to any of the others which use the fewest number of swaps. Once all pairwise partial vector products have completed on all computers for a particular permutation, that permutation is removed from the global list of unused permutations which is shared across all computers.

4.5. MPI optimization: load balancing

The MPI algorithm described in the previous section grows increasingly inefficient when the pairwise partial vector product operations are not load-balanced across all computers. In Algorithm 5, there is a `continue` statement which can dramatically reduce the runtime when the subgraph studied by one computer is less dense than that on another computer. When the entire adjacency matrix fits on all computers, this is easily addressed by identifying a random graph automorphism by performing a Fisher–Yates shuffle [47] of labels. This allows each computer to choose unique random pairs, though it introduces a small amount of overhead.

On the other hand, if the adjacency matrix must be split among multiple computers, load balancing is much more difficult. If we suppose that in a four-computer setup the `for` loops on two computers finish long before those on the other two, it would make sense for the idle computers to perform possible memory swaps and resume work rather than remain idle. The dynamic design in

Fig. 3 addresses this flaw by permitting transfers to be performed independently until all operations are finished.

The primary difficulty with such a design is that for this problem, MPI calls require all computers to listen and respond, even if they do not participate in a particular data transfer. The reason for this is that the temporary storage used for an individual swap is spread across all computers to minimize overhead and balance memory requirements. Therefore, each computer uses two POSIX threads: a master thread listens and responds to all MPI calls, and also monitors whether the computer is active or idle with respect to action calculations, while a slave thread performs all tasks related to those calculations. A shared flag variable indicates the active/idle status on each computer.

As opposed to static MPI action algorithm, where whole permutations are fundamental, buffer pairs are fundamental in the load-balanced implementation. This means there is a list of unused pairs as well as a list of pairs available for trading, i.e., those pairs on idle computers. When two computers are both idle, they check to see if a buffer swap would give either an unused pair, and if so they perform a swap. After a swap to an unused pair, the computer moves back from an idle to an active status.

5. Simulations and scaling evaluations

5.1. Spacetime region considered

In benchmarking experiments, we choose to study a $(1+1)$-dimensional compact region of de Sitter spacetime. The de Sitter manifold is one of the three maximally symmetric solutions to Einstein's equations, and it is well-studied because its spherical foliation has compact spatial slices (i.e., no contributing boundary terms), constant curvature everywhere, and most importantly, a non-zero value for the action. We study a region bounded by some constant conformal time η_0 so that the majority of elements, which lay near the minimal and maximal spatial hypersurfaces, are connected to each other in a bipartite-like graph.

The $(1+1)$-dimensional de Sitter spacetime using the spherical foliation is defined by the metric

$$ds^2 = \sec^2\eta(-d\eta^2 + d\theta^2)\,, \tag{9}$$

and volume element $dV = \sec^2\eta\, d\eta\, d\theta$. This foliation of the de Sitter manifold has compact spatial slices, meaning the manifold has no timelike boundaries. Elements are sampled using the probability distributions $\rho(\eta|\eta_0) = \sec^2\eta/\tan\eta_0$ and $\rho(\theta) = 1/2\pi$, so that $\eta \in [-\eta_0, \eta_0]$ and $\theta \in [0, 2\pi)$. Finally, the form of (9) indicates elements are timelike-separated when $d\theta^2 < d\eta^2$, i.e., $\pi - |\pi - |\theta_1 - \theta_2|| < |\eta_1 - \eta_2|$ for two particular elements with coordinates (η_1, θ_1) and (η_2, θ_2). This condition is used in the CUDA kernel which constructs the causal matrix in the asynchronous GPU linking algorithm.

We expect the precision of the results to improve with the graph size, so we study the convergence over the range $N \in [2^{10}, 2^{17}]$ in these experiments. Larger graph sizes are typically used to study higher-dimensional spacetimes and, therefore, will not be considered here. We choose a cutoff $\eta_0 = 0.5$ in particular because for η_0 too small we begin to see a flat Minkowski manifold, whereas for η_0 too large, a larger N is needed for convergence since the discreteness scale $l = \sqrt{V/N}$ is larger.

5.2. Convergence and running times

Initial experiments conducted to validate the BD action show that the interval abundance distribution takes the form as in manifold-like causal sets (versus in Kleitman–Rothschild partial orders) [48], and that the mean begins to converge to the EH action around $N \gtrsim 2^{14}$, Fig. 4. The Ricci curvature for the constant-curvature de Sitter manifold is given by $R = d(d+1)$ so that the EH action is simply

$$S_{EH} = \frac{d(d+1)}{2}V(\eta_0) = 4\pi\tan\eta_0\,. \tag{10}$$

We note that the standard deviation σ_S in Fig. 4 (right) increases as $O(\sqrt{N})$ because we have chosen to keep the smearing parameter ε fixed as N increases, which is the more common practice, but if we had instead chosen $\varepsilon \to \varepsilon/\sqrt{N}$, then σ_S would go 0 as $N \to \infty$ [24]. While normally one would need to consider the Gibbons–Hawking–York boundary terms which contribute to the total gravitational action, it is known that spacelike boundaries do not contribute to the BD action [25] and the codimension-2 boundary does not contribute, since the BD action violates the Lorentzian Gauss–Bonnet Theorem [49,50].

These calculations are extremely efficient when the GPU is used for element linking and AVX is used on top of OpenMP to find the action (Fig. 5). The GPU and AVX optimizations offer nearly a 1000x speedup compared to the naive linking and action algorithms, which in turn allows us to study larger causal sets in the same amount of time. The decreased performance of the naive implementation of the linking algorithm, shown in the first panel of Fig. 5, is indicative of the extra overhead required to generate sparse edge lists for both future and past relations. There is a minimal speedup from using asynchronous CUDA calls because the memory transfer time is already much smaller than the kernel execution time.

5.3. Scaling: Amdahl's and Gustafson's laws

We analyze how Algorithm 5 performs as a function of the number of CPU cores (n_c) to show both strong and weak scaling properties (Fig. 6). Amdahl's Law, which measures strong scaling, describes speedup as a function of the number of cores at a fixed problem size [51]. Since no real problem may be infinitely subdivided, and some finite portion of any algorithm is serial, such as cache transfers, we expect at some finite number of cores the speedup will no longer substantially increase when more cores are added. In particular, strong scaling is important for Monte Carlo experiments, where the action must be calculated many thousands of times for smaller causal sets. We find, remarkably, a superlinear speedup when the number of cores is a power of two and hyperthreading is disabled, shown by the solid lines. The dashed lines in Fig. 6 indicate the use of 28, 32, and 56 logical cores on dual 14-core processors.

We also measure the weak scaling, described by Gustafson's Law [52], which tells how runtime varies when the problem size N^3 per processor P is constant (Fig. 6(right)). This is widely considered to be a more accurate measure of scaling, since we typically limit our experiments by the runtime and not by the problem size. Weak scaling is most relevant for convergence tests, where the action of extremely large graphs must be studied in a reasonable amount of time. Our results show nearly perfect weak scaling, again deviating when the number of cores is not a power of two or hyperthreading is enabled. We get slightly higher runtimes overall when more computers are used for two reasons: the computers are connected via a 10Gb TCP/IP cable rather than Infiniband and the load imbalance becomes more apparent as more computers are used. Since the curves have a nearly constant upward shift, we believe the likely explanation is the high MPI latency. For each data point in these experiments, we "warm up" the code by running the algorithm three times, and then record the smallest of the next five runtimes. All experiments were conducted using dual Intel Xeon E5-2680v4 processors running at 2.4 GHz on a Redhat 6.3 operating system with 512 GB RAM, and code was compiled with nvcc 8.0.61 and linked with g++/mpiCC 4.8.1 with Level 3 optimizations enabled.

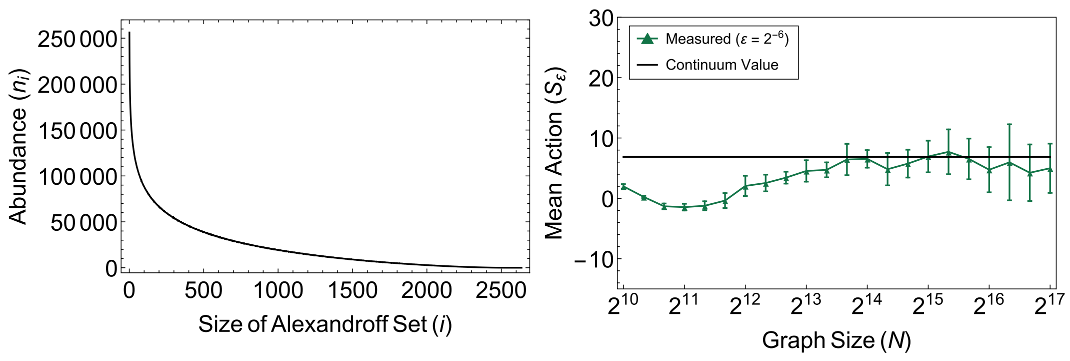

Fig. 4. The action in (1 + 1)-dimensional de Sitter spacetime. The left panel shows the interval abundance distribution for a (1 + 1)-dimensional de Sitter slab with $N = 2^{15}$ and $\eta_0 = 0.5$. The right panel shows the BD action (green) converging toward the EH action (black) as the graph size increases. We take a symmetric temporal cutoff $\eta_0 = \pm 0.5$ and a small smearing parameter $\varepsilon = 2^{-6} \ll 1$ so the onset of convergence appears as early as possible. Remarkably, the terms in the series (5) are several orders of magnitude larger than the continuum result $S \approx 6.865$, yet the standard deviation about the mean is quite small, shown by the error bars in the second panel. The error increases with the graph size because the smearing parameter ε is fixed while the discreteness scale $l = \sqrt{V/N}$ decreases. All data shown are averaged over ten graphs. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

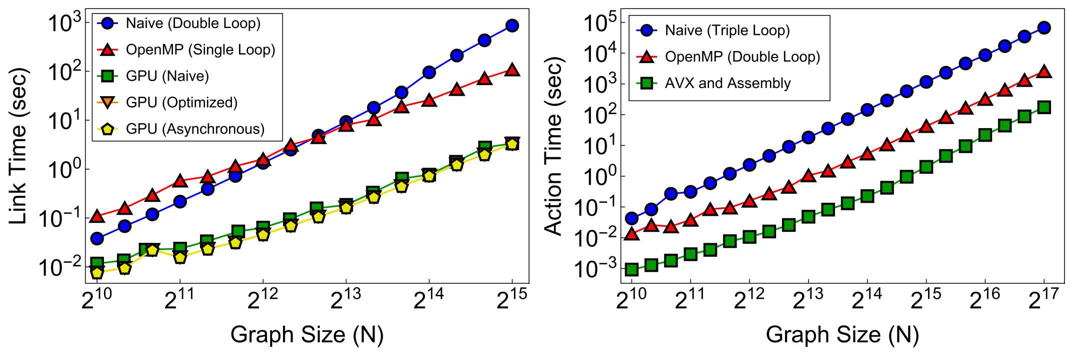

Fig. 5. Performance of the linking and action algorithms. We benchmark the $O(N^2)$ linking algorithm (left) and the $O(N^3)$ action algorithm (right) over a wide range of graph sizes. The left panel shows moving from a sparse (blue) to a dense (red) representation improves the scaling of the linking algorithm, though it can still take several minutes to generate causal sets of modest size. When the NVIDIA K80m GPU is used, we find a dramatic speedup compared to the original implementation, which allows us to generate much larger causal sets in the same amount of time. We find the three variations of the GPU algorithm (green, orange, yellow) provide nearly identical run times. The right panel shows the benefits of using both OpenMP and AVX instructions to parallelize. The optimal OpenMP scheduling scheme varies according to the problem size, though in general a static schedule is best, since it has the least overhead. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

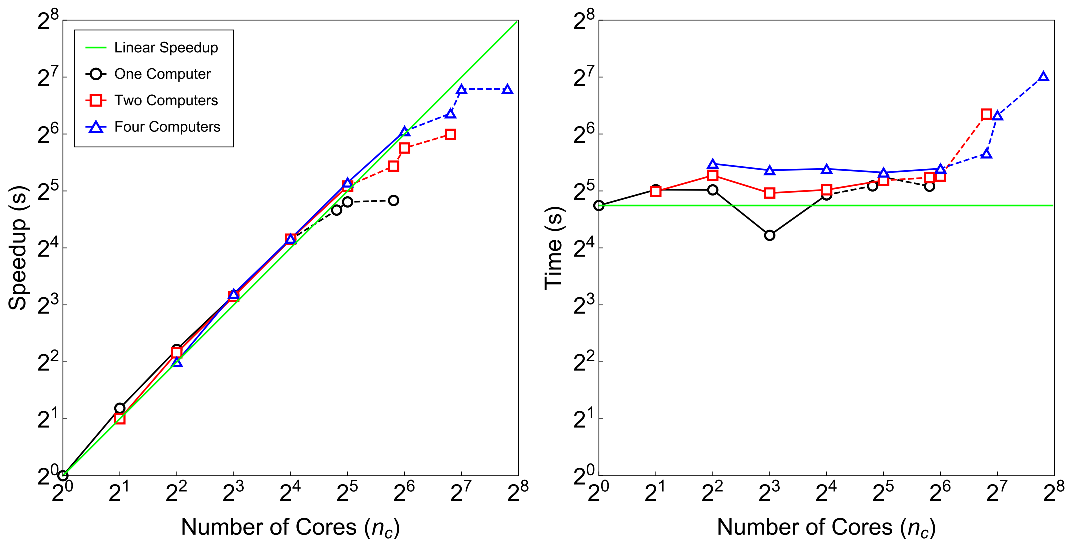

Fig. 6. Strong and weak scaling of the action algorithm. The action algorithm exhibits nearly perfect strong and weak scaling, shown by the straight green lines in each panel. The `for` loop in Algorithm 5 is parallelized using OpenMP, while the partial inner product is parallelized using AVX. When multiple computers are used, pairs identified by the loop are evenly distributed among all computers. We find the best speedups when the total number of cores used is a power of two and hyperthreading is disabled (solid lines). When we use all 28 physical cores, or we use 32 or 56 logical cores in our dual Xeon E5-2680v4 CPUs, we find a modest increase in speedup (dashed lines). In the right panel, the runtime should remain constant while the number of processors is increased as long as the amount of work per processor remains fixed. The constant increase in runtime when more computers are added is likely due to a high MPI communication latency over a 10Gb TCP/IP network. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

6. Conclusions

By using low-level optimization techniques which take advantage of modern CPU and GPU architectures, we have shown it is possible to reduce runtimes for causal set action experiments by a factor of 1000. We used OpenMP to generate the element coordinates in parallel in $O(N)$ time and used the GPU to link elements much faster than with OpenMP. By tiling the adjacency matrix and balancing the amount of work each CUDA thread performs with the physical cache sizes and memory accesses, we allowed the GPU to generate causal sets of size $N \gtrsim 2^{20}$ in just a few hours. We developed the efficient and compact `FastBitset` data structure to overcome limitations imposed by other similar data structures, and implemented ultra-efficient intersection, bit counting, and inner product methods using assembly in Algorithms 2, 3 and 5. The MPI algorithms described in Sections 4.4 and 4.5 provide a rigorous protocol for asynchronous information exchange in the most efficient way when the adjacency matrix is too large to fit on a single computer. Finally, we demonstrated superlinear scaling of the action algorithm with the number of CPU cores, indicating that the code is well-suited to run in its current form on large computer clusters.

Acknowledgments

We thank J. Chartrand, D. Kaeli, C. Orsini, D. Rideout, N. Roy, S. Surya, and P. Whitford for useful discussions and suggestions. This work was supported by National Science Foundation grants CNS-1442999, CNS-1441828, and IIS-1741355.

References

[1] L. Bombelli, J. Lee, D. Meyer, R.D. Sorkin, Phys. Rev. Lett. 59 (1987) 521–524.
[2] R.D. Sorkin, Causal sets: Discrete gravity, in: A. Gomberoff, D. Marolf (Eds.), Lectures on Quantum Gravity, Springer US, Boston, MA, 2005, pp. 305–327.
[3] R. Sorkin, in: J. D'Olivo, E. Nahmad-Achar, M. Rosenbaum, M. Ryan, L. Urrutia, F. Zertuche (Eds.), Relativity and Gravitation, World Scientific, 1990, pp. 150–173.
[4] S. Surya, Class. Quantum Grav. 29 (2012) 132001.
[5] D. Benincasa, F. Dowker, Phys. Rev. Lett. 104 (2010) 181301.
[6] G. Brightwell, J. Henson, S. Surya, Class. Quantum Grav. 25 (2008) 105025.
[7] P. Wallden, J. Phys. Conf. Ser. 222 (2010) 012053.
[8] S. Surya, Directions in causal set quantum gravity, (2011) arXiv:1103.6272.
[9] S.W. Hawking, A.R. King, P.J. McCarthy, J. Math. Phys. 17 (2) (1976) 174–181.
[10] D.B. Malament, J. Math. Phys. 18 (1977) 1399.
[11] M. Penrose, Random Geometric Graphs, Oxford University Press, Oxford, 2003.
[12] P. Winkler, Order 1 (1985) 317–331.
[13] L. Glaser, D. O'Connor, S. Surya, Class. Quantum Grav. 35 (2018) 045006.
[14] S. Surya, Priv. Commun. (2017).
[15] S. Surya, Making Quantum Gravity Computable, 2017.
[16] J. Myrheim, Statistical Geometry (1978) CERN TH-2538.
[17] D. Meyer, The Dimension of Causal Sets, (Ph.D. thesis), Massachusetts Institute of Technology, 1989.
[18] D. Rideout, P. Wallden, J. Phys. Conf. Ser. 174 (2009) 012017.
[19] F. Dowker, L. Glaser, Class. Quantum Grav. 30 (2013) 195016.
[20] L. Glaser, Class. Quantum Grav. 31 (2014) 095007.
[21] S. Aslanbeigi, M. Saravani, R. Sorkin, J. High Energy Phys. 2014 (2014) 24.
[22] A. Belenchia, D. Benincasa, F. Dowker, Class. Quantum Grav. 33 (2016) 245018.
[23] D. Benincasa, F. Dowker, B. Schmitzer, Class. Quantum Grav. 28 (2011) 105018.
[24] D. Benincasa, The Action of a Causal Set, (Ph.D. thesis), Imperial College London, 2013.
[25] M. Buck, F. Dowker, I. Jubb, S. Surya, Class. Quantum Grav. 32 (2015) 205004.
[26] P. de Maupertuis, Mem. l'Acad. Sci. Paris (1744) 417–426.
[27] I. Gelfand, S. Fomin, Calculus of Variations, Prentice-Hall, New Jersey, 1963.
[28] R. Wald, General Relativity, University of Chicago Press, Chicago, 1984.
[29] T. Regge, Nuovo Cim. 19 (1961) 558–571.
[30] K. Wilson, Phys. Rev. D 10 (1974) 2445.
[31] J. Henson, D. Rideout, R. Sorkin, S. Surya, Exp. Math. 26 (2017) 253–266.
[32] G. Allen, T. Goodale, F. Löffler, D. Rideout, E. Schnetter, E. Seidel, 11th IEEE/ACM International Conference on Grid Computing, 2010.
[33] D. Kleitman, B. Rothschild, Trans. Amer. Math. Soc. 205 (1975) 205–220.
[34] J. Griffiths, J. Podolský, Exact Space–Times in Einstein's General Relativity, Cambridge University Press, New York, 2009.
[35] OpenMP Architecture Review Board, OpenMP application program interface version 3.1, 2011. http://www.openmp.org.
[36] G. Brightwell, P. Winkler, Order 8 (1991) 225–242.
[37] N. Sato, W. Tinney, IEEE Trans. Power Appar. Syst. 82 (1963) 944–950.
[38] W. Tinney, J. Walker, Proc. IEEE 55 (1967) 1801–1809.
[39] NVIDIA Corporation, CUDA C programming guide, 2017. http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf. Version PG-02829-001_v8.0, (Accessed 11 July 2017).
[40] H. Wong, M. Papadopolou, M. Sadooghi-Alvandi, A. Moshovos, 2010 IEEE International Symposium on Performance Analysis of Systems Software, ISPASS, 2010, pp. 235–246.
[41] Boost Community, Boost C++ libraries, 2017. http://www.boost.org.
[42] Intel Corporation, Intel intrinsics guide, 2017. http://software.intel.com/sites/landingpage/IntrinsicsGuide. (Accessed 11 July 2017).
[43] J. Weidendorfer, Intel Core Microarchitecture, x86 Processor Family, Springer, US, Boston, MA, 2011, pp. 936–944.
[44] A. Fog, The Microarchitecture of Intel, AMD and VIA CPUs, 2017, http://www.agner.org/optimize/microarchitecture.pdf, (Accessed 22 January 2018).
[45] A. Fog, Instruction Tables, 2017, http://www.agner.org/optimize/instruction_tables.pdf, (Accessed 25 September 2017).
[46] W. Muła, N. Kurz, D. Lemire, Comput. J. (2017) 1–10.
[47] R. Fisher, F. Yates, Statistical Tables for Biological, Agricultural, and Medical Research, third ed., Oliver & Boyd, London, 1948.
[48] L. Glaser, S. Surya, Phys. Rev. D 88 (2013) 124026.
[49] P. Law, Rocky Mountain J. Math. 22 (1992) 1365–1383.
[50] D. Benincasa, J. Phys. Conf. Ser. 306 (2011) 012040.
[51] G. Amdahl, Proceedings of the 1967, Spring Joint Computer Conference, AFIPS '67 (Spring), ACM, New York, NY, USA, 1967, pp. 483–485.
[52] J. Gustafson, Commun. ACM 31 (5) (1988) 532–533.