International Journal of Networking and Computing – www.ijnc.org
ISSN 2185-2839 (print) ISSN 2185-2847 (online)
Volume 7, Number 1, pages 2–28, January 2017

Entropy of Parallel Execution and Communication

Ernesto Gomez
School of Engineering and Computer Science
CSU San Bernardino
San Bernardino, CA, 92407, USA

Zongqi "Ritchie" Cai
Department of Electrical and Computer Engineering
Baylor University
Waco, TX, 76798, USA

Keith Evan Schubert
Department of Electrical and Computer Engineering
Baylor University
Waco, TX, 76798, USA

Received: July 25, 2016
Revised: November 26, 2016
Accepted: December 21, 2016
Communicated by Susumu Matsumae

Abstract

We propose a definition of parallel state, derive a phase space from this state, and calculate the entropy of states and full executions using combinatorial analysis. A main contribution of this work is the introduction of an experimentally measurable phase space, which we then use to analyze execution states, ensembles of states, and ensembles of complete executions. We show that the entropy analysis reveals both expected and unexpected features of execution, and application of principal component analysis shows capability to extract execution details at the level of individual process states, as well as reveal hardware properties such as network or memory communications.

Keywords: entropy, phase space, parallel state, interprocess communication

1 Introduction

The need to understand parallel execution via static and dynamic traces has been recognized for at least 30 years. Visualization of trace data helps programmers, compiler designers, and hardware designers improve their products. Dozens of tools exist to help visualize how a parallel code executes that can be used to visualize dynamic execution (gprof, openpat, etc) or static execution (cflow, doxygen, codeviz, etc). Dynamic tools monitor actual executions, and can use program instrumentation, timed sampling, random sampling, event based sampling, and even hardware support. Static tools use formal and statistical methods to examine possible executions, and do not rely on runtime data. One thing that continues to come up from all of these tools and their related papers, is the problem of the size of the data and the difficulty visualizing the data. We extend a simple method to visualize either static or dynamic trace data using phase space and entropy, initially introduced in [20]. There is a long history to entropy in computation and visualization of execution traces even involving entropy in the analysis, so the following sections provide background and distinguish the present work from others in the area using similar concepts in different ways.

1.1 Entropy in Computation

Entropy has a long history in computation, starting with information theory and proceeding through image processing and very large scale integration (VLSI). Brandt and Pompe in [6] argue that permutation is the correct measure for time series.

Entropy has been applied to execution traces in [22], where the complexity of a trace is estimated by the entropy rather than the size of the trace. Since entropy is related to information, entropy yields a good measure of the available information in the trace. This is significantly different than how entropy is used in this paper however. In this work entropy is used to analyze the state of the run and system as well as the available parallelism, thus applying entropy to understand the system and program rather than assess the data. Both are useful and even complementary.

Entropy has been applied to examining the complexity of graphs in [4, 5, 23]. The analysis used also included principal component analysis(PCA), and involved calculating time complexity and traces, but is completely different in its meaning and application. The time complexity and traces are for converting graphs, PCA is used to examine the graphs themselves. This paper does not deal with graph theory, though there is obviously a strong relation between program execution and graph theory.

Multidimensional entropy has been used in networking [46] for network traffic, anomaly detection, and spectrum usage. [46] used real and synthetic data from various probabilistic models to analyze and validate the models and detect unusual patterns, which can be used for prediction. Their work is significantly different than this paper, but is supporting in that it shows network behavior is accurately measurable from time series data.

Entropy has also been applied to artificial intelligence (AI), see [54], to measure the complexity, adaptation, and evolution of an AI process. There are obvious similarities in measuring how a program run is going, however this work is broader, in that it applies to any piece of code. Additionally, this work centers on measuring performance of a parallel run, rather than how an AI program evolves. A natural extension of the use of entropy for AI is the use of entropy for EEG signals, see [36, 45], where it was shown to predict seizures in the brain analogous to how we use it to predict problems in program runs. In [26], transfer entropy between two time series is used to reveal the network structure of the cerebral cortex, similar to how entropy of an ensemble of time series is used to exam the computational and communication structure in our work.

Entropy has been applied to VLSI (very large scale integration) to handle complexity of signals and circuits, see [37, 51], for everything from reliable memory [52] to reversible logic [38, 28] to decoupling capacitor placement [56]. Entropy is also not surprisingly used in a number of energy and power related ways, such as partitioning for power estimation [14], VLSI floorplanning for temperature reduction [8], and low-power computing [29, 11]. Link entropy is a standard measure of evaluating network connections, see [55]. Information entropy bounds were used to first compare neural networks [47] then to design VLSI neural networks [13]. Entropy provide a method for analyzing algorithmic complexity and performance in hardware and providing upper bounds that are useful in design. In this paper, the ability of entropy to capture algorithmic complexity and performance are used to examine general parallel software running on multiprocessors. In the present paper, as in many VLSI papers, network performance is revealed by entropy but in this case the network performance data is mixed with computational data providing an extra layer of difficulty. Further, we show that performance of an algorithm correlates with entropy, since energy spent by an algorithm is proportional to the work done, and thus the time spent performing the work by the code.

Most recently, Garland et al, have used entropy in time series of cache memory and processor load to explore determinism and predictability of an execution, see [15, 16]. They note that as entropy increases, determinism decreases, which corresponds to our observations as well. The inverse relation between entropy and determinism was shown to be true for dynamic systems by Pesin, see [44], and numerous results show computers [41] and cellular automata [50, 7] can be modeled by dynamic systems. Our work is concerned with different data - parallel execution traces and network accesses as opposed to instructions per cycle (IPC) and cache misses - and different methods - principal component analysis and visualization versus dynamical systems.

1.2 Visualization Tools

One of the early tools for visualizing message passing parallel programs was ParaGraph, see [24]. Currently there are many popular tools to visualize parallel traces, including Pajé [10], Ocelot [3], Vampir [33], Vite [9], Projections [31], and Viva [48]. Dosimont, et al note in [12] that current tools for visualizing large execution traces use Gantt charts, timelines, task profiles, communication matrices, and treemap/topology, all of which have limitations due to screen resolution and rendering artifacts. Sigovan et al in [49] use animation to try and handle the difficulty of using Gantt charts and kiviat diagrams for large systems, which we handle with PCA and entropy. Mohror and Karavanic in [40] use pattern-based methods to for performance analysis, to reduce size and handle the complexity of the data. They propose a way of partitioning and aggregation, which they evaluate using divergence and entropy, but do not apply entropy to the visualization. Similarly, Lamarche-Perrin et al in [35] use entropy as a performance measure for visualization, but not to actually visualize the data. Pagano et al in [42] note the problem of trace visualization scalability exists for embedded systems. The state of the art report on Performance Visualization from EuroVis 2014, see [27], shows no use of PCA or entropy to visualize performance, showing that the ideas presented in this paper are novel contributions.

Process discovery is an unsupervised learning task used on trace logs to construct a process model, however due to the size and complexity of the data, techniques like clustering [53] are used to help mine the data. While [53] does use entropy to evaluate the cluster sets, they note that it can't be used in real-life settings since the ex ante classes of behavior are unknown. While the present work uses entropy to analyze trace logs in an unsupervised way, it is not for the purpose of clustering or process modelling, and thus is fundamentally different.

2 Program State

To define phase space, we start from execution models of Hoare [25] and Apt and Olderog [2]; a set of sequential recursive communicating processes. Unlike Apt and Olderog, we will not restrict ourselves to interleaved execution, but like them we will not consider nested parallelism. In this work we restrict ourselves to a static process set, membership of which is known at each process and which does not change during execution.

We now address process state. Based on Apt and Olderog's Verification of sequential and Concurrent Programs (VSCP) [2], a proper state is defined by the contents of variables accessible to the process, and state changes when any one of the variables is updated. Defining state change in this way is not true to a modern processor with multiple instruction issue and out of order execution. The instruction stream as given by code defines a correct execution order; what actually happens is one of many possible actual executions guaranteed to produce the same result. VSCP [2] is concerned with program correctness, it's definition works for that purpose.

We are concerned with measuring real execution, so we prefer a definition from computer architecture: we take state to be given by contents of the program counter plus memory accessible to the process. This architectural state is too detailed for practical use in a phase space; however most memory content is not necessary to track execution. We may still identify variables to specify state (for example, a finite automaton) or control variables (loop indices or predicates) in specific problems.

For a general approach we define execution in terms of sets of statements that always execute together, in whatever order. Such a set is well known in compiler theory as a basic block [1], and useful in phase tracking and granular studies of code [32]. Because we wish to study communicating concurrent programs, we extend the standard definition so a communication or synchronization statement always ends the current block (we introduced this in [17]).

A basic block can be regarded as a function on current memory contents, which yields the memory contents on input to the next block. Standard compiler analysis serves to find which variables are actually read and modified inside the block [1]. State change happens when a process completes execution of one basic block and starts the next one.

Since basic blocks are defined by control structures, any change in execution order by the hardware will generally not cross block boundaries. Basic blocks can be arranged in a directed Control Flow Graph (CFG) which denotes allowed transitions between blocks. Control flow from high level code is preserved by translation to low level, so the CFG is useful as a language-independent representation of a program.

We model sequential program execution as walk through a CFG, represented as a list of basic blocks starting in a designated start block and ending at a designated end block, indexed by step number [2], [1], [17], [19]. Given the initial state of memory, its content at any step is obtained by ordered application of the basic block functions at each preceding step. Barring explicit non-deterministic code, the list of prior blocks uniquely determines state of memory at any given step. We use this fact in noting that, given a list of blocks up to a state, the memory state is fixed and so we can mostly avoid the need to explicitly represent it in the phase space.

For a process q, state is:

$$\sigma_{q,j} = (x_j, M_j)$$

where j denotes execution step, x_j the block executed, and M_j the contents of memory at entry to x_j. A transition $\sigma_{q,j} \to \sigma_{q,j+1}$ occurs when the process applies transformations $x_j(M_j) \to M_{j+1}$ defined in x_j and enters x_{j+1}. Each process follows a path $p_q = s_q \to^* e_q$ through a control flow graph $G =< V, A, s, e >$(V is the set of nodes denoting basic blocks, A is the set of directed arcs between nodes, s and e are designated start and end blocks); so p_q describes the execution. The sequence $s \to^* e$ defines the transformation of memory, by repeated application of $x_j(M_j) \to M_{j+1}$.

We now define (following [2]) a parallel state as the set of the states of each concurrent process at the same external clock time. Given a set Γ of P processes, all of which start in a basic block labeled s and end in a basic block labeled e, the state of a parallel execution is the set of the individual process states that exist concurrently :

$$S_{\Gamma,J} = \{\sigma_{i,j}\}$$

This is a total state of all processes; $i \in \Gamma$ is an ordered set of process numbers and $j \in J$ is a multi-index which denotes the step number of each process in Γ. We may define a partial state involving a subset of processes $S_{I,J}$, with $I \subset \Gamma$. A transition $S_{P,I} \to S_{Q,J}$ occurs when at least one, and up to every $q \in P \cap Q$ transitions to its respective next state. For total states we have $S_{\Gamma,I} \to S_{\Gamma,J}$ and all processes are in the intersection. If there are $n = |P \cap Q|$ processes in a particular parallel transition there are $n!$ possible next states when we consider all possible combinations of processes that may transition.

We describe a parallel execution in total states $S_{\Gamma,s} \to^* S_{\Gamma,e}$ in [17]. We here consider a parallel execution in the Single Program, Multiple Data stream SPMD [30] model, which features task parallelism with replicated code. This restricted model with P processes of the same code is for convenience in experiments, and does not preclude extending our methods to a more general model.

3 Phase Space

In physics, a phase space is simply the space of all possible states that can occur. Phase space does not have to include everything about the system, just whatever is significant for the behavior under study. For example, the standard phase space for the study of a classical gas of N particles has $6N$ dimensions, three momentum and three position per molecule but does not represent the internal energy and angular momentum states [43]. Similarly, we restrict our definition to serve our purpose, while allowing later extension. For a parallel execution trace, everything would include the space of all possible combinations program traces and data values. Since full data traces are usually impractical, let alone full traces of the data values associated with a program, we choose to represent parallel states (of basic blocks) rather than entire executions, because we wish to expose details inside execution.

We will consider the phase space as the space of all possible combinations of the available program traces for all code in the parallel execution. Program trace data for each sequential execution that makes up the parallel ensemble thus defines a single axis of the phase space. Thus a program with 10 threads defines a 10-dimensional phase space. We propose an N dimensional space such that each dimension maps an independent system feature; with P processes we pick the basic block from each process state. A single point in the phase space denotes a set of at least 1 block number per process to describe a configuration. If we wish to represent selected variables we may need an additional dimension per variable, or we may replace block number with a representative value. Our phase space represents total states of Γ processes, but does not distinguish partial states of $G \subset \Gamma$; such states may be identified as needed by selecting a sub-space.

The CFG restricts the possible successor states for any given point S, and the successor is not always adjacent to S in the phase space. However, because blocks are numbered by textual position, a higher numbered state usually represents a more advanced point in execution; this is the case in all our current examples.

3.1 Representation parametric in time

It is convenient to label the start block s with the number 0, number all other blocks (except the end block) by their order in the program text, and number the end block e as the highest block. We may also take a specific variable that is set inside a basic block and characterizes its action and use it instead of the basic block label (this approach is limited to integer variables with known range that can be used to characterize execution of the block). This can reveal more detailed process information than the basic block label, but is code and problem dependent.

For SPMD execution, since all processes have the same code, the phase space is bounded by a P-dimensional hypercube with sides of length e (without the restriction to SPMD we have a hyper-rectangle). Each state $S_{\Gamma,J}$ is represented by a P-tuple of block numbers x_j from the state.

The path in phase space is a parametric graph of the progress of individual processes, with time as the parameter. For an execution that terminates normally, the path begins on the corner labeled $(0, 0, \ldots, 0)$ of the hypercube and ends at $(e, e, \ldots, e)$ which is the opposite corner. A lockstep execution would appear as a straight line between start and end vertices. Task parallel executions in which processes may concurrently choose to execute different subsets of blocks will show specific deviation from the center line when processes execute such regions. Less restricted executions would have states away from the diagonal, up to anywhere in the space.

3.2 Explicit inclusion of time in the phase space

Since states are identified by an external time t, for every state $S_{\Gamma,J}$ we could add t to the state. A set of states from the same execution can be ordered by t and a parameter τ can be used to denote a count of states traversed up to t. Either parameter could be used as coordinate for a time dimension.

However the information in $S_{\Gamma,J} = \{(x_j, M_j)\}$ is clearly sufficient for entropy, which is not dependent on history. The content of x_j, M_j is sufficient for our needs; indices are labels we use to keep external track of the states. In this view, t or τ would be external labels to distinguish different states; their values are not known internally to the execution, so they are not in the phase space.

3.3 Displaying P/P+1 dimensional data

We display the phase space as a (hyper) cube standing on a diagonal, with the line from $(0, 0, \ldots, 0)$ to $(e, e, \ldots, e)$ displayed vertically. We use a 3D perspective graph displayed on a plane, and can rotate the graph and view it from different angles.

If the phase space is for 3 processes, this looks like a standard cube standing on one vertex, with x y and z axes departing diagonally from point $(0, 0, 0)$ at angles of $\pi/3$ with the horizontal plane. The projections of the 3 axes x,y,z on the horizontal are separated by angles of $\frac{2\pi}{3}$; we display x at an angle of 0 as standard.

For dimensions $P > 3$, we follow the same scheme, separating the axes by $\frac{2\pi}{P}$. We calculate the (x,y,z) coordinates for unit vectors u_i lying along each of the dimensions we wish to display. The point $X_p = (x_i, \ldots x_p)$ in P dimensions is placed in 3D at: $X_3 = \sum_i^P (u_i x_i)$ where the u_i are 3D vectors, the x_i are scalar integer values and the resulting X_3 is a 3D vector from the origin.

The visualization is contained in a volume that looks like two P sided pyramids joined at the base, aligned so the axis of symmetry is perpendicular to the horizontal XY plane and touching the origin (for large P the pyramids look like two cones joined at the base).

We find this display to be intuitive. If processes advance in lockstep through the same code then every state should look like $(k, k, \ldots, k)$ where k is a block id, and the execution plot should display all points on a vertical line. Barrier synchronizations involving all processes should also display on the center line, and points should cluster near synchronizations as processes wait for others to arrive. Excursions away from the center line reflect processes advancing at different rates or subsets of processes doing different tasks concurrently.

No attempt has been made to compensate for distortion, so scales on graphs are only usable for comparison between graphs of the same type and do not give a true value of the P-dimensional distances and positions that the graphs are generated from.

3.4 Principal components of higher dimensional data

Principal component analysis (PCA) has been discovered many times in different fields, and hence goes under a variety of names. The fundamental idea behind PCA is to find a new set of bases such that the data in the direction of the first basis has its widest variance, and the next highest variance of the data is in the direction of the second basis, and so on. One straightforward way to perform PCA is to center the data and then perform a singular value decomposition (SVD). In MatLab this can be done by either of the built in commands **pca()** or the deprecated **princomp()**. In either case, the '*score*' output is the data projected in the directions of the components. Assuming the run data for each process is in columns, and different times in rows, then the *score* matrix will have the principal component of all runs in the first column, which will show the basic behavior of the code on the system. The second column will similarly contain the second principal component, which is how the different nodes were likely to be at that particular instant. For a machine running in lockstep, the second and higher components will be zero, as only one component will be needed to describe them all, since they would be identical. PCA can also return the percent of the data variance that each component accounts for; in MatLab this is in the output, '*explained*' of **pca()**.

By plotting the principal components in order, we can guarantee that no matrix transformation that is plotted can display more variance in the data. Plotting the first principal component against time or sample number will capture more of the diversity of what is happening than any other plotting of the state versus time or sample. Similarly plotting the first two singular values will better represent the fullness of what is occurring in the program trace or phase space than any other 2-D plot. Thus a 3-D plot of the first three principal components will capture more of the variability in the run than any other 3D plot. This property of maximum variance in the first components is related to the idea of entropy, which quantifies variability, and is described in the next section.

4 Entropy

Entropy has a long history in Computer Science, starting with the foundational work of Claude Shannon on entropy of information. Entropy is standardly used in information theory, imaging, and data analysis, but interestingly has not been applied to understand program traces. Shannon entropy is defined as:

$$-\sum p_i \log_2(p_i)$$

where p_i is the probability that some state or event i occurs (Note that Shannon entropy is distinct from the related Shannon information formula).

To compute p_i for a state, we note that in our case each process has identical code, therefore two states with the same mix of basic blocks, even with different process numbers, can be thought equivalent - (they are identical in the terms of our phase space). For P processes and N basic blocks in the code, the number of equivalent states are the possible permutations of the blocks composing the state, given by the total possible permutations divided by the permutations of each subset in the state with the same block number [39]:

$$p_i = \frac{P!}{N^P \prod (b_k)!}$$

where b_k with $k \leq P$ are subsets of processes with the same basic block in the state (e.g. in a state of P=10 with 3 processes in block 1, 5 in block 2 and 2 in block 3, the b_k would be 3!, 5! and 2!) . N^p is the hyper volume of the phase space used as a normalization factor. Permutations are lowest for a state where all processes are in the same block as in a barrier. States with the same mix of basic blocks can be considered a priori to have the same probability, and therefore the number of possible combinations is a proxy for the probability of the state.

We considered an alternate approach of estimating probability from observed densities in experimental data. The exponential increase of hyper volume with dimensionality of a space made it infeasible to gather enough data for useful measured density values. For a sufficiently large number of processes the entropy of the execution can be calculated analogous to entropy of an image, see [21]. Perform a histogram on each time period, with each bin corresponding to the blocks, then normalize it by dividing each bin by the total number of processes. The histogram is now an approximation of the probability density function (pdf) and each bin contains an estimate of the probability, p_i, of being in that bin, thus Shannon entropy can then be directly calculated on the normalized histogram.

5 Experiments

5.1 Simplex

Baseline code was an implementation of an amoeba downhill simplex algorithm, which uses N+1 processes, each assigned to a vertex of the simplex. Test function to minimize was an N-dimensional hyperboloid with known minimum, and the start point for the algorithm was chosen to get convergence in 20 iterations. This gives an example of code with a barrier.

Note that, since the simplex must be a solid in the problem space, it has one more dimension than the problem; a problem of dimensionality N will require N+1 processes in our algorithm and the dimensionality of the phase space will be at least N+1.

Our tests were written in C with MPI (Message Passing Interface) - a generally accepted standard for message passing communication, and SOS (Streams, Overlapping and Shortcutting) - an asynchronous communication library introduced by one of us in [17], which is written on top of MPI. We used the MPICH implementation of MPI from Argonne National Laboratory.

In the parallelization, each of the N+1 vertices of the simplex was assigned to a different processes, and each run was started at the same position. Each process then modified the position of its vertex to produce a new simplex. In the SOS and MPI runs, these simplexes were then compared in a reduction operation, and the best was broadcast to all process to begin the next iteration. A

Table 1: Basic blocks with id numbers for amoeba simplex code. Note - it always does block 4, then does either blocks 5 and 6, or blocks 7 and 8.

Block #	Description
1	copy initial arrays simplex (coordinate array for vertices) fval (value of function at vertices)
2	start of while loop loop over fval array to determine maximum and minimum node value initialize test points array for computing new node
3	calculate projection from ME through opposite base finds test point, evaluates FUN
4	just before choice point
5	if result(3) was good - extends projection, evaluates FUN
6	chooses between two projections
7	shrink simplex - evaluates FUN dim+1 times at each vertex
8	end of shrink simplex
9	loops over fval to find new best, worst points
10	checks for convergence global reduction op - find best value and node that has it prints current state of computation
11	broadcast simplex array from node with best value
12	recalculate fval array by evaluating FUN dim+1 times (all vertices) end of loop
13	converged, print results outside loop

convergence test was performed at each process to determine when to stop (halt was forced at 20 iterations for the "none" case).

Four variants of the code were used - labeled "none", "guided", "sos" and "mpi".

The "sos" code uses MPI asynchronous sends and receives which underlie a parallel C (PC, [34]) compiler supported by one of us. The "mpi" code uses standard MPI broadcast and reduction operations; these are blocking code, and have a synchronizing effect similar to a barrier. The "none" code had no inter-process communication, and was hard coded to stop in 20 iterations. The "guided" code did not communicate, but had access to a matrix with the data computed by the MPI code at each iteration; the "guided" code therefore used the same partially computed data from MPI execution and converged on the same computational path.

The basic blocks of the program are in Table 1. Note: each FUN call is $(loadfactor) * (dim) * 9$ double precision floating point ops (3 mult, 6 add). The loadfactor is 10^6 for runs on Xeon systems.

Each data set included 5 runs of 20 iterations each, for each of the four code variants ("mpi", "sos", "none" and "guided"). Data sets are shown for a 17 dimensional phase space and a 32 dimensional phase space (respectively 16 and 31 dimensional problems).

17 dimensional runs were on AKEK, a parallel research cluster at California State University San Bernardino, with 7 nodes of 12 (24 with hyperthreading) Intel Xeon 2.4 GHz cores each, using MPICH over 1 Gigabit Ethernet or 20 Gigabit Infiniband. Processes were scheduled in round-robin mode on 6 nodes.

32 dimensional runs (and all prefix runs) were on TARDIS, a parallel research cluster at Baylor University, with 5 nodes of 24 (48 with hyperthreading) Intel Xeon 2.7 GHz cores, using MPICH over either shared memory or 40 Gigabit Infiniband.

5.2 Dining Philosophers

We implement a variant of the "dining philosophers" problem In this problem a group of philosophers are sitting at a table; there is one fork between each par of philosophers with each philosopher initially

Table 2: Basic blocks with id numbers for prefix-mpi.C code

Block #	Description
0	init local variables find midpoint of current process subset
1	Recursive Case: create new MPI communicator depending on upper/lower half of subset find size and id in new communicator (value of ME resets for each communicator)
2	ME in bottom half: call pref with lower half communicator
3	ME in top half: call pref with upper half communicator
4	Return from recursive call:
	Free subset communicator: ME resets to value in caller if ME == exact middle, ME value is top of lower half
5	broadcast top prefix from top process in lower set if(ME) in top half, add broadcast value
6	BASE CASE: run delay routine
7	return to caller with prefix value

holding the fork on his left. The philosophers need to get two forks to eat, which will allow them to think about some problem. This is a well known example of mutual exclusion, which can exhibit starvation, live-lock, and even deadlock. We allowed three basic states: EAT, ASK, THINK.

Transitions THINK⟷ASK, ASK →EAT, EAT →THINK are allowed. A philosopher in THINK state is oblivious to the world and any forks he holds may be taken by his neighbors. A philosopher in ASK will hold on to any fork and will try to take forks he needs from his neighbors. A philosopher must have two forks to move from ASK to EAT, is otherwise only allowed transition to THINK.

To prevent deadlock, Philosophers were restricted to a maximum of 30 turns in THINK, 10 in ASK and 10 in EAT, so eventually are forced back into the THINK state where they do not hold on to forks. In addition there was a .3 probability that a philosopher would change state on each iteration.

We tested on AKEK and on ERIS (a quad-core Intel machine), with 9 processes in each case and 1000 iterations. The program ensures absence of deadlock, but makes no effort to ensure fairness; live-lock or starvation is unlikely but possible. We tested a symmetric case with all philosophers on an equal basis, and simulated a live-lock situation in an asymmetric case by preventing philosopher 5 from acquiring 2 forks so he could never EAT. We did two symmetric and two asymmetric runs on each system.

5.3 Parallel prefix summation

Prefix summation is a standard problem used in parallel programming courses. In this case, parallel prefix was coded as a recursive program in MPI. The basic parts of the program are listed in Table 2. All blocks are in the recursively called routine:

```
double pref(double y, int rank, int size, MPI_Comm procset);
```

5.4 Instrumenting the code

We wanted a full trace of the system so we decided not to go with a random or timed sampling. It would be possible to record a total state by interrupting all processes at a specific time and inspecting memory, but this would be too intrusive for use in recording an execution. We choose instead to record the sequence of basic blocks and times (x_j, t_j) at each process, and use these to reconstruct a total execution after the fact. We could have recorded calls instead of basic blocks, but we felt basic blocks were more general and the results should be the same.

In the downhill simplex code, we numbered basic blocks sequentially from 1 to 12, giving a hypercube of side 11. Locations 2-10 were inside the minimization loop. All communication was in blocks 9 to 11.

In the dining philosopher code we treated the loop interior as one block and identified the philosopher states using a 4 bit code. The high two bits denote THINK=00, ASK=01 and EAT=10, and the low two bits are set to the bitwise OR of 10 for the left fork and 01 for the right fork, with 00 denoting no forks. This gives us 9 states numbered 0 to 8 (ignoring the lower two bits of EAT).

At entry to each block at each process we print "time block pid" to a file numbered by (pid) in its name, creating one file per process. Here "name" is the program name, "pid" is process id from 0 to P-1, b is block number and t is time in microseconds to $10\mu s$ resolution. These files are then concatenated and piped to the UNIX "sort" command, which creates a single file "name+ident.srt" sorted on time and with "time block pid" on each line. (Multiple runs of the same program on same data and system are concatenated for convenience but not sorted so the different executions are preserved in the file by noting the time dropping back to 0).

Timer resolution is constrained by available hardware. Some states including blocks that take very little execution time will not be resolved by the present system, so it will generate an approximate view.

5.5 Reconstructing parallel state

We read the data into a matrix of P+1 columns. Each row represents a single parallel state; the first P columns contain the basic block with column number as process id, column P+1 is the time at at which the state was observed. The first row is arbitrarily assumed to be time 0 with all processes at the start block. After the first row, each entry in the raw data file is matched to the time of the current row. If the time is the same, the block is written to the corresponding column. If the time has changed, a new row is created with the new time, the block is written to the corresponding column and all other columns are filled in from the previous row. This continues for every single process state in the raw data file, and the resulting matrix has M rows corresponding to the number of states found.

It is possible that two successive entries in raw data for the same process will display the same time and different blocks; this can happen for blocks that execute in times shorter than the time resolution of the present system. We find this to be rare based on spot comparisons between raw data and the generated matrix.

6 Experimental runs

6.1 Amoeba Simplex

Each of the graphs displays results for 5 runs of 20 loop iterations and 17 concurrent processes for the amoeba program. 3D graphs have standard XYZ coordinates. Data from AKEK,11 April 2015.

Scales represent the 3D projection of 17D euclidean distance from the origin - given that the program starts in block 1 and ends in block 12, these distances are roughly indicative of where a given state is in the code.

Graphs with no communications are similar to "sos", see fig. 2 in distribution, but lacking clumps at top and bottom. Only the "guided" case is shown, see Figure 3.

6.2 Dining philosophers

Each graph represents one run of 1000 iterations with 9 concurrent processes on 6 nodes, Data from AKEK January 2016. Scales represent the 3D projection of 9D euclidean distance from the origin. Each dimension takes on values from 0 to 8, with higher values denoting a philosopher in ASK or EAT with more forks.

Figure 1: Synchronous MPI

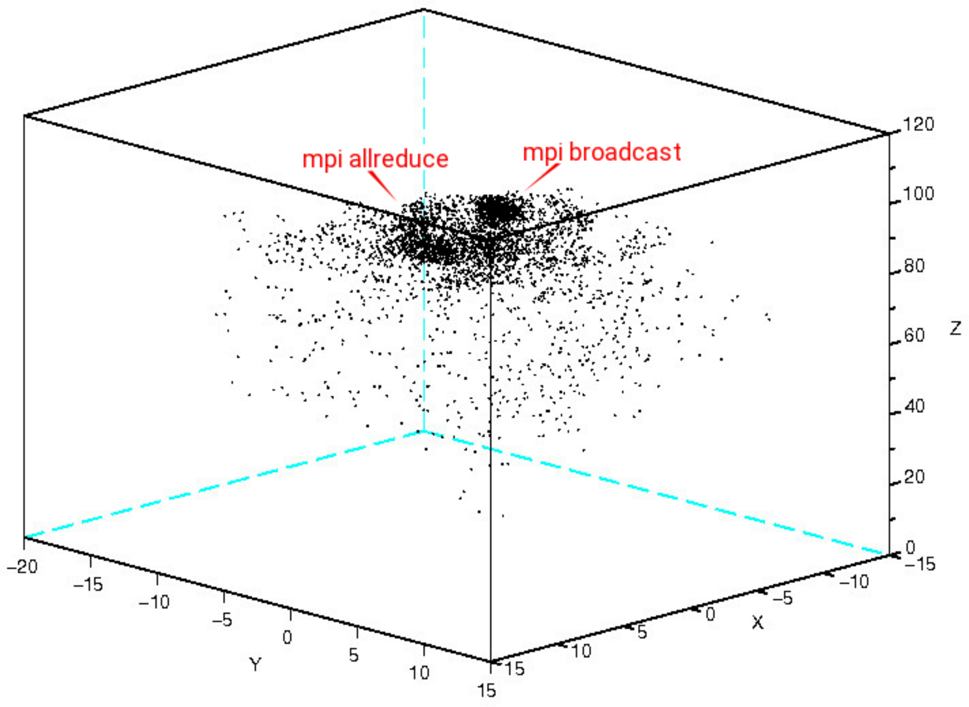


Figure 2: Asynchronous SOS

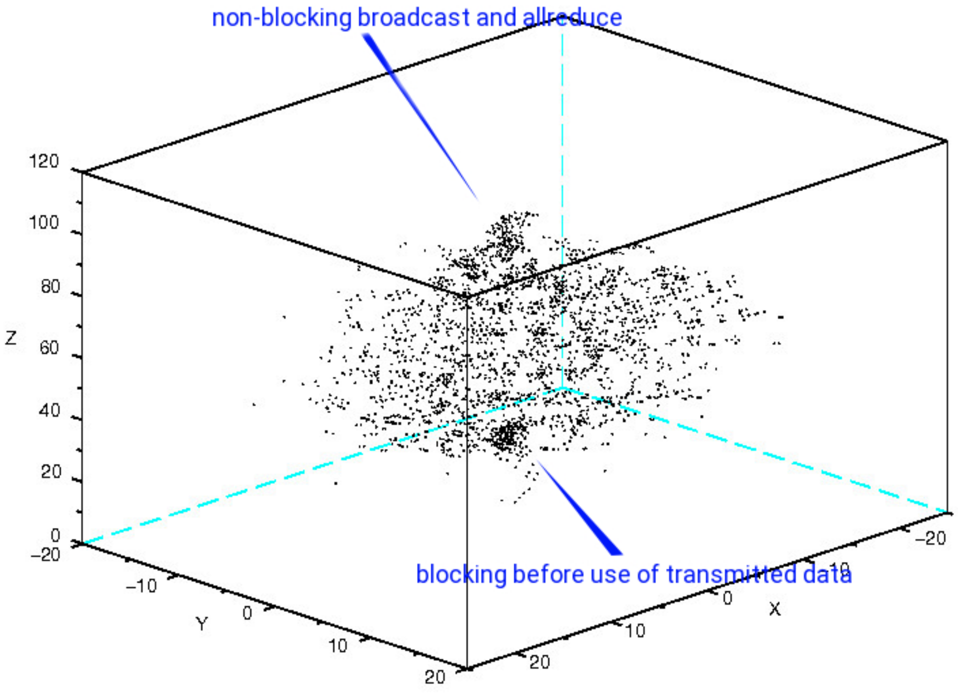


Figure 3: No communication guided

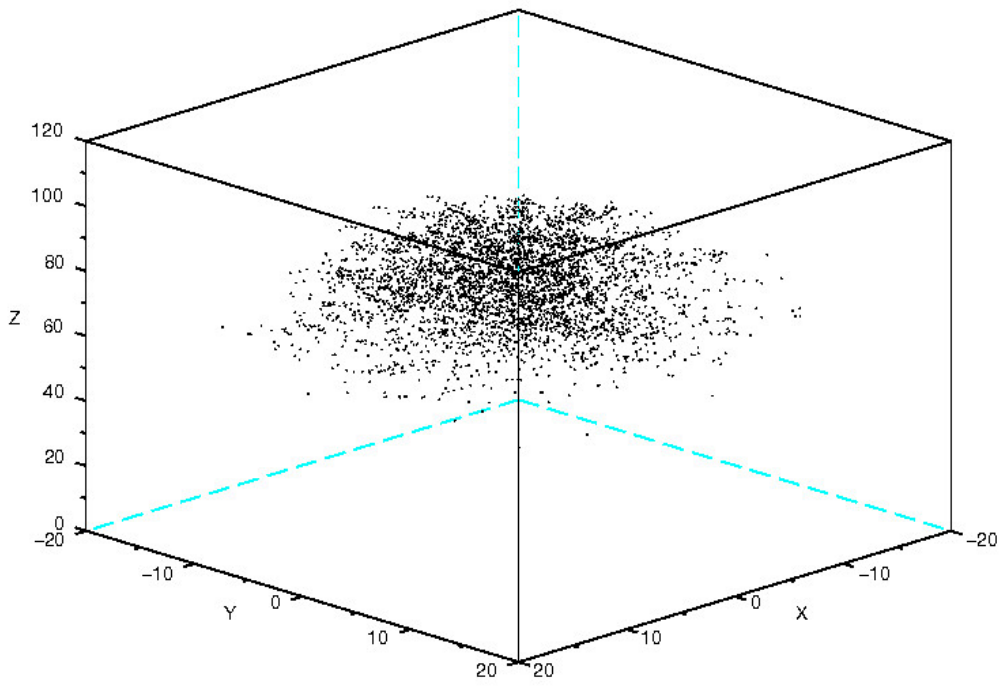


Figure 4: Symmetric dining philosophers

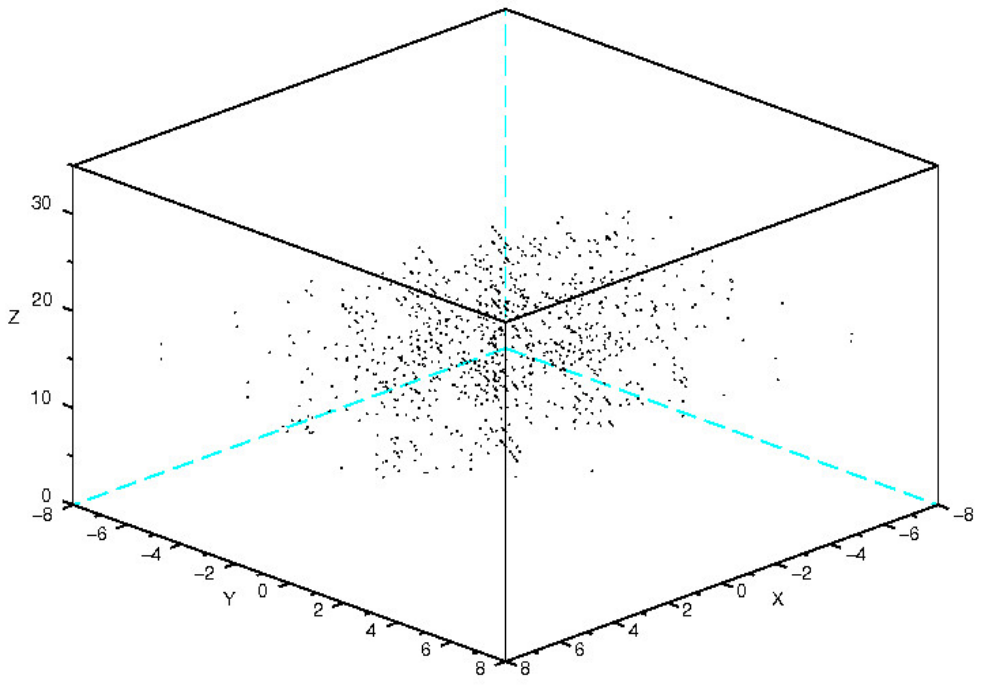


Figure 5: Live-lock dining philosophers

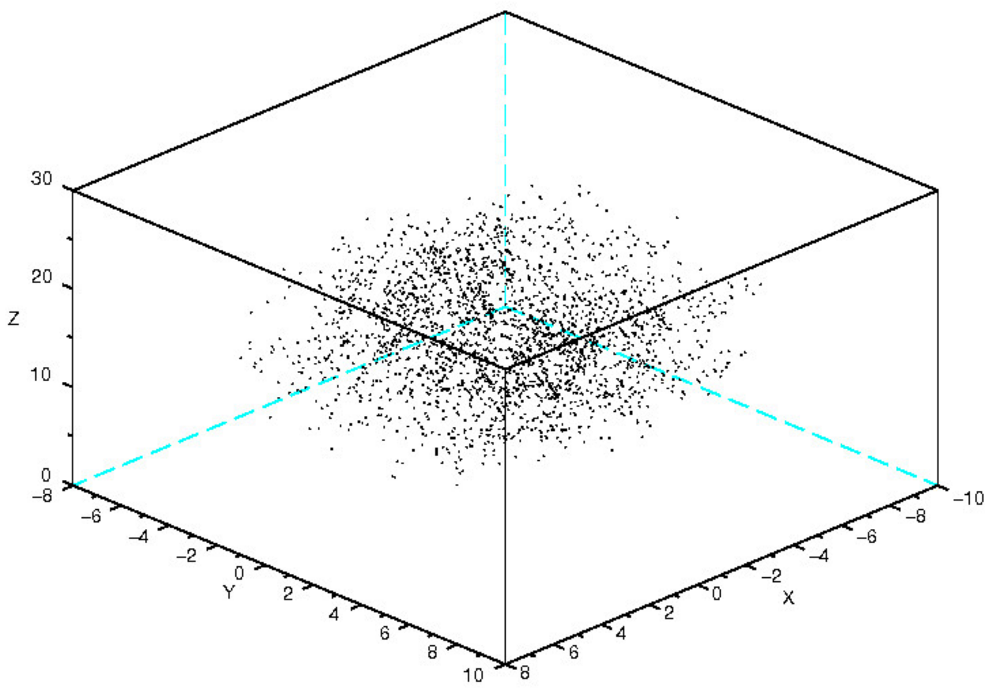
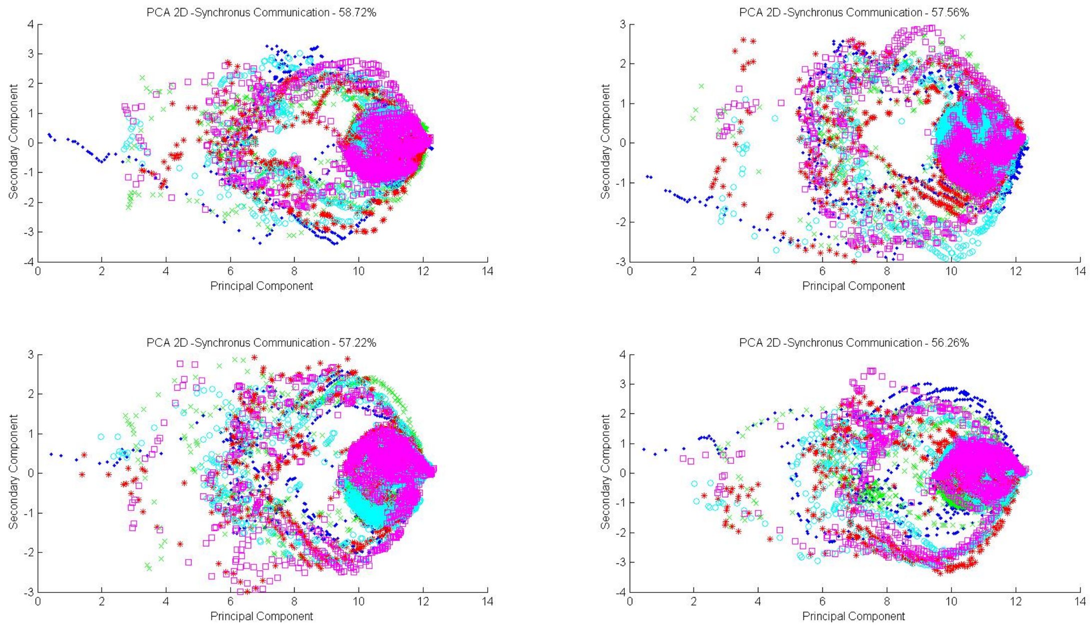


Figure 6: First two principal components of synchronous (MPI) amoeba simplex code. The four sub-figures show four executions, each of which had 5 runs. The colors and shapes in the sub-figures show the run (blue dots are first, green 'x' are second, red asterix are third, cyan circles are fourth, and magenta squares are fifth). The size of the markers indicate the time; smaller markers are earlier, larger markers are later in the run. The percentage is the average percent of the data's variation that is seen in the graph.



6.3 Non-determinism

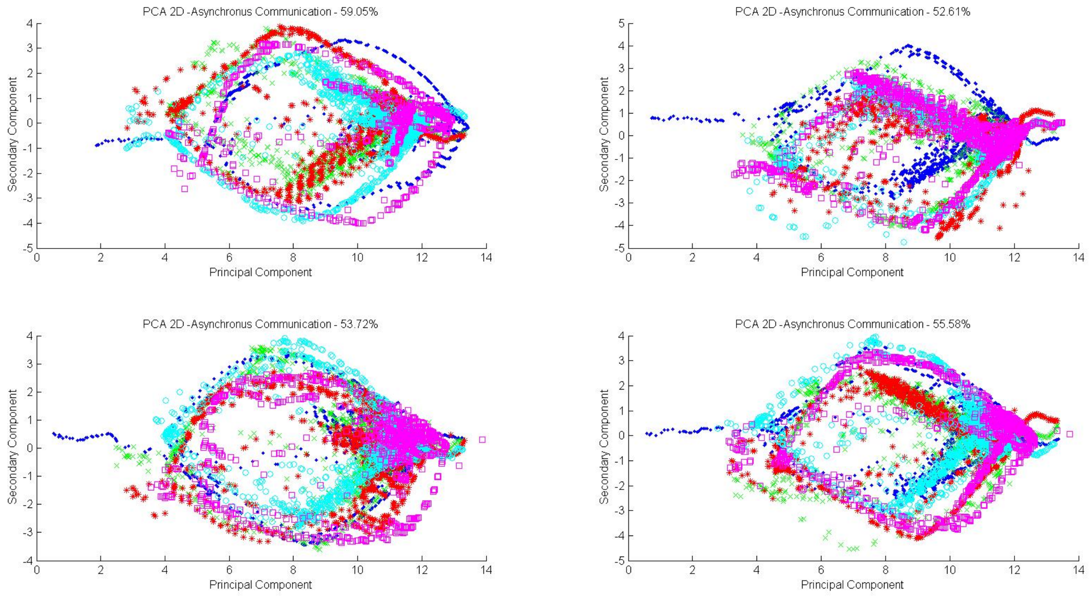
Execution was on a lightly loaded homogeneous cluster only running operating system functionality and our test code, with each process on a separate core.

We expected some excursions from a vertical line of synchronous code, but we also expected that the general trend would leave most states close to the center line; we find little indication of such a trend. The ensemble is closer to the center line near the top and bottom of the graphs, but this seems to correspond to the shape of the phase space more than to anything else. The distribution of next states seems akin to a random walk. Particularly the "SOS" Figure 2, and no communication executions Figure 3 seem to expand into most regions of the phase space.

The graph of the first two principal components for the synchronous amoeba simplex, Figure 6 shows that each run has key features, but demonstrates significant differences also. Two features particularly stand out: the large clump on the right discussed in Section 6.4, and the empty center. As discussed in Section 3.4, the principal component shows the biggest overall variation, which corresponds to the typical process execution. The second principal component is how the processes in the ensemble vary at that point. The empty center of the graphs from around block 6 till block 9 is caused by the two separate routes: execution either follows block 4 $\rightarrow$ block 5 $\rightarrow$ block 6 $\rightarrow$ block 9 or block 4 $\rightarrow$ block 7 $\rightarrow$ block 8 $\rightarrow$ block 9. The spreading of the two routes is caused by variation in the composition of states in the ensemble. The asynchronous communication used in the sos execution, Figure 7, causes the separation of the two paths through the code ($4\rightarrow5\rightarrow6\rightarrow9$ or $4\rightarrow7\rightarrow8\rightarrow9$) to be even more distinct since data is available when needed.

In dining philosophers, we have a single code block where a philosopher thinks, tries to get forks from neighbors or eats. We chose a representation that displays internal state of each philosopher. Since the acquisition of forks requires mutual exclusion, adjacent philosophers are likely to be in

Figure 7: First two principal components of asynchronous (SOS) amoeba simplex code. The four sub-figures show four executions, each of which had 5 runs. The colors and shapes in the sub-figures show the run (blue dots are first, green 'x' are second, red asterix are third, cyan circles are fourth, and magenta squares are fifth). The size of the markers indicate the time; smaller markers are earlier, larger markers are later in the run. The percentage is the average percent of the data's variation that is seen in the graph.



different states. Dining philosopher phase space graphs both display the expected spread away from center; the missing high value states in process 5 (no states numbered 8 for EAT or 7 for ASK with 2 forks) show up in an apparent reduction of symmetry for the live-lock graph Figure 5.

6.4 Effects of synchronization

Synchronization is carried out by making processes wait, particularly true for barriers affecting all processes. Our test code has two global collective communications - an allreduce followed by a broadcast. Both require every process to wait for data, so they act like barriers. Both communications appear near the end of the loop, so we expect clumping near the top.

The MPI graph Figure 1 shows very strong clumping - processes reach the last blocks of the loop and wait there for others. This is seen even more strongly in Figure 6. SOS uses non-blocking communications, but it will nevertheless block as needed to preserve data dependencies. Any synchronizing effects would appear later in the code, at the point of data use. Further, there should be less clumping because some processes will actually have received the data before they need to use it, and so never block. We have less clumping at the top than for MPI, but also have some clumping at the bottom of the SOS graphs, see Figure 2 - the data of the broadcast at the top of the loop is not used until the start of the next iteration, and so any blocking shows up there. By examining the plot of the first two principal components, see Figure 7, this is even more clear. The clump around block 10 through block 12 has been significantly spread out, and since block 4 is common to both paths the more evenly spread out communication block appears as a mild concentration point just before it is used.

Consider the graphs of the first principal component of the synchronous (MPI) amoeba simplex runs versus sample number in Figure 8. All runs took the same number of samples regardless of run time, samples indicate position in the run, not the time it took to get there. Note first that each graph is composed of an initial line corresponding to the setup followed by 20 'v' patterns corresponding to the 20 iterations. Near the start of the calculation there are three larger 'v' shaped iterations which correspond to the amoeba simplex attempting to grow to find the right size. The remaining calculations are much quicker so the communications dominate. The 'v' pattern is caused by the processes all at the barrier, then when they pass the barrier, not all receive the message at the same time. The first ones to get the message jump down to basic block 4, causing a slight dip, since most are still waiting for the message. As more states get the message, the first ones have moved back up, and the graph proceeds toward the bottom of the 'v'. Once all states have exited the barrier the graph, is at the bottom of the 'v', but by this time the first ones have already reached the next barrier, since for later iterations the computation is small. As more processes finish the graph proceeds up the 'v' till all are waiting and the process repeats. This is what causes the strong clump in the two component graph.

Now consider Figure 9, which is essentially the same graphs, except for the asynchronous communication case. There are still 20 'v' shaped segments, and the three corresponding to pseudopod growth are distinctly visible. The key difference is that the 'v' shapes are taller and more spread out. This is what causes the spreading of the clump between basic blocks 10 and 12. In this case the principal component varies from basic block 8 till 12. This corresponds to higher variability in the execution, also known as entropy. The greater entropy allows more flexibility in scheduling the instructions and a more efficient run, resulting in the asynchronous execution taking around 35ms to 72ms versus around 280ms to 300ms for the synchronous case.

A logical question is how large of an influence is communication latency and conflict? To examine this we stored all the data sent in an execution in a file, and allowed the processes to just read the answers from RAM, thus reducing a network transmission to a memory access. The result is in Figure 10. The 20 'v' shapes of the 20 iterations can be seen, but they are significantly distorted, since the processes are essentially independent. The independence can be clearly seen in how the graphs all drift upward at the end, since some processes finish early and enter basic block 13. As more and more do, the slower ones still show their 'v' shape but it is tilted up, since they are averaged with more and more processes that are done. This is why there are more jagged shapes to the earlier 'v' shapes - there are faster nodes creating what looks like noise, but is actually just a

Figure 8: Principal component of synchronous (MPI) amoeba simplex code versus sample point. The four sub-figures show four executions, each of which had 5 runs. The colors in the sub-figures show the run (blue lines are first, green lines are second, red lines are third, cyan lines are fourth, and magenta lines are fifth). The percentage is the average percent of the data's variation that is seen in the graph.

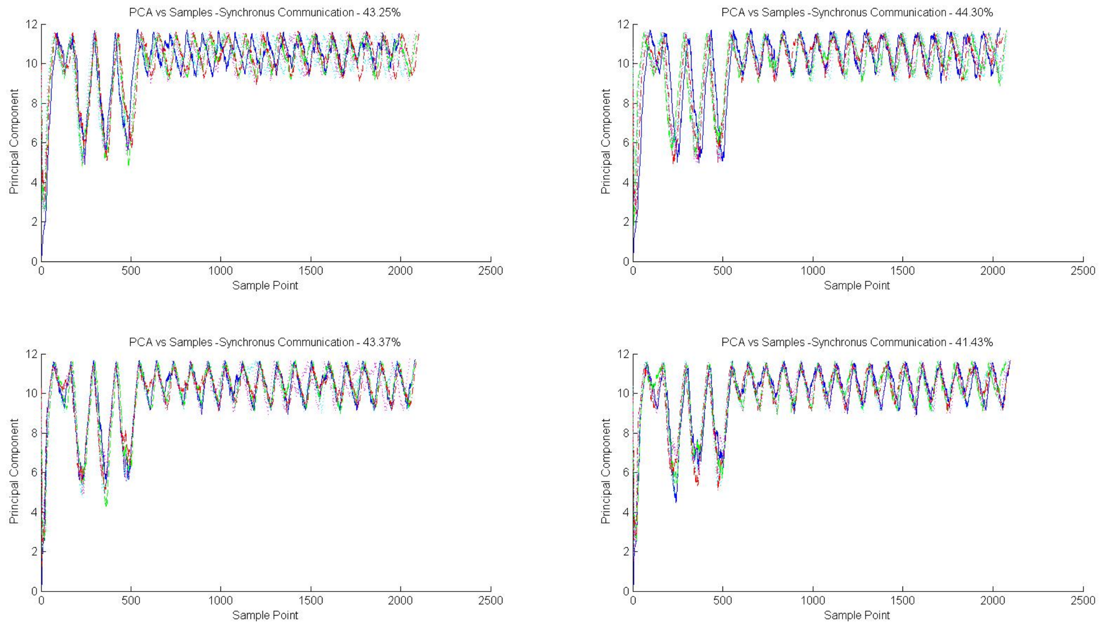


Figure 9: Principal component of asynchronous (SOS) amoeba simplex code versus sample point. The four sub-figures show four executions, each of which had 5 runs. The colors in the sub-figures show the run (blue lines are first, green lines are second, red lines are third, cyan lines are fourth, and magenta lines are fifth). The percentage is the average percent of the data's variation that is seen in the graph.

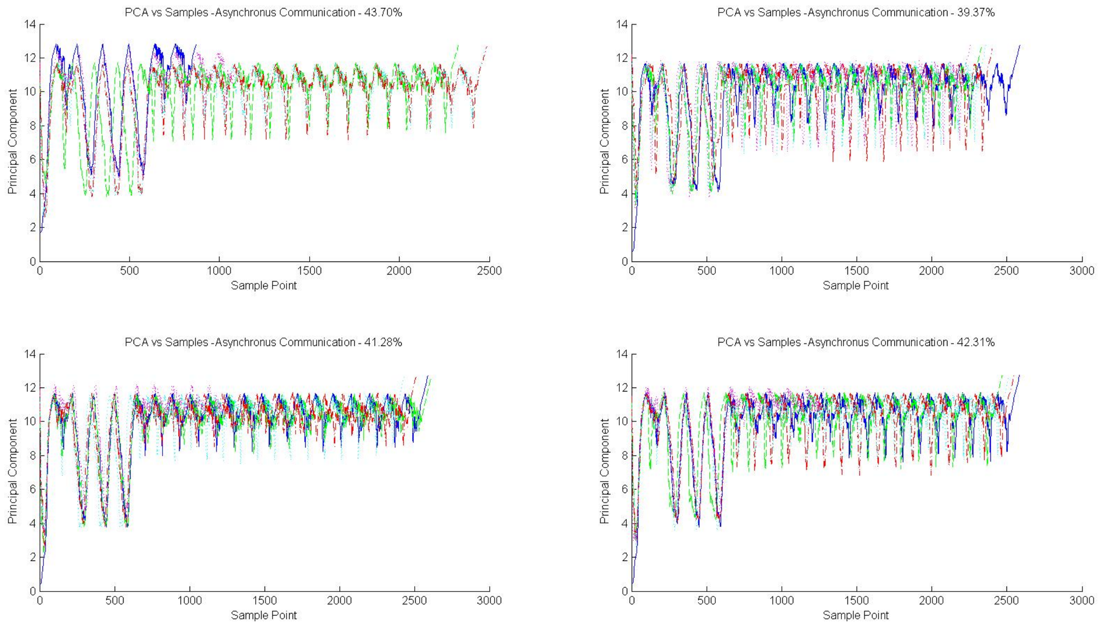
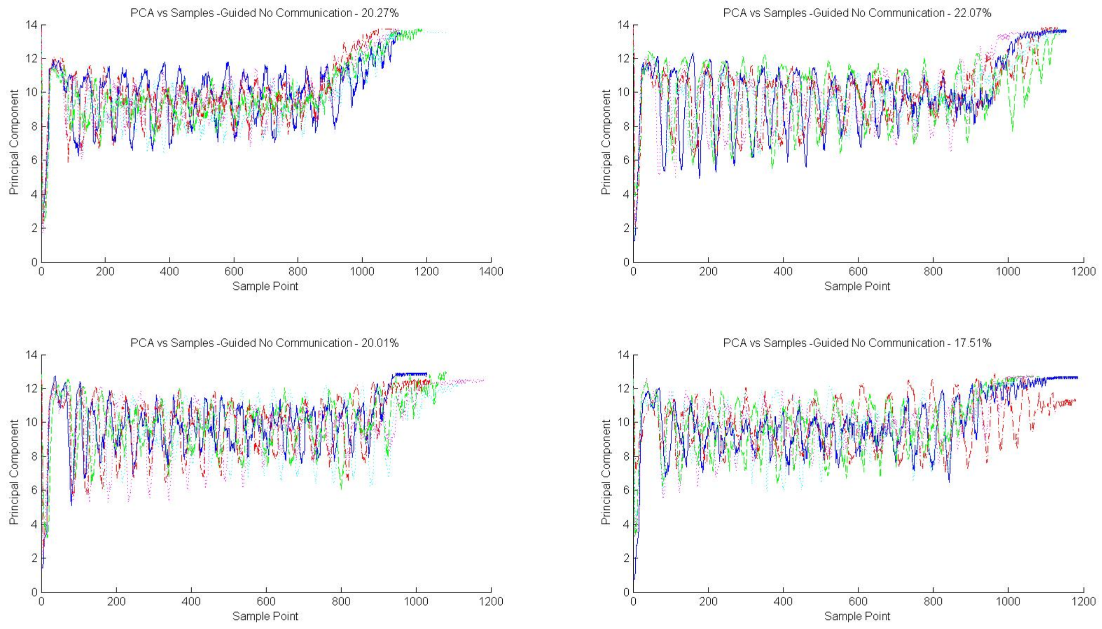


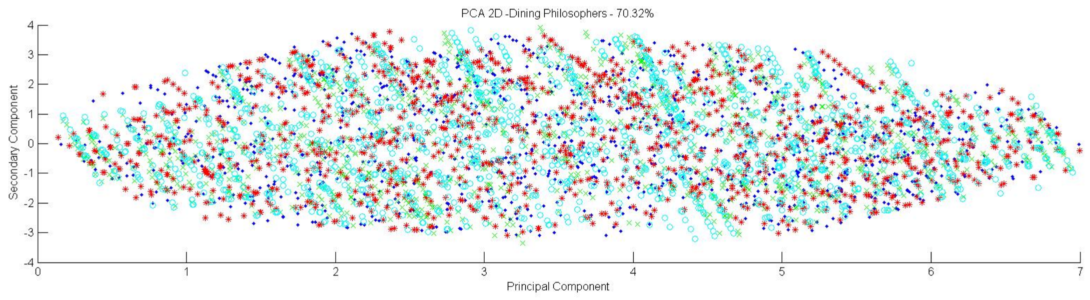
Figure 10: Principal component of guided no communication amoeba simplex code versus sample point. The four sub-figures show four executions, each of which had 5 runs. The colors in the sub-figures show the run (blue lines are first, green lines are second, red lines are third, cyan lines are fourth, and magenta lines are fifth). The percentage is the average percent of the data's variation that is seen in the graph.

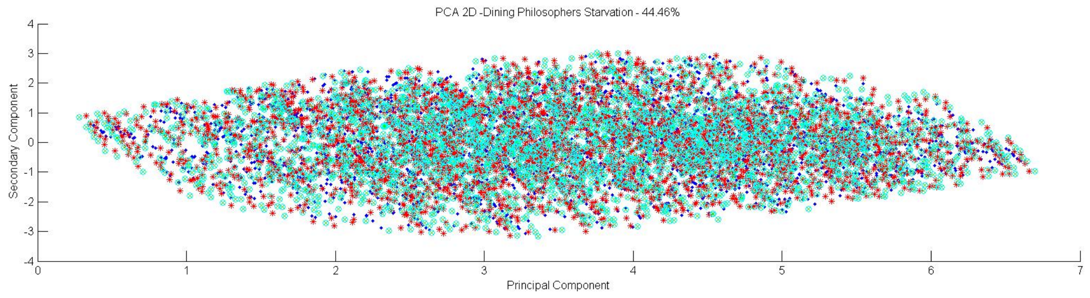


few fast nodes out of sync with the bulk of the others. The effect of 32 out of sync processes creates a beat frequency, which can be seen in the pattern of the variable heights of the 'v'. The Tardis cluster at Baylor uses Intel Xeon E5-2697 cores, which have Intel's Turbo Boost 2.0 technology, allowing the cores to adjust their frequencies opportunistically based on a variety of factors such as local temperature. It is thus unlikely that the cores will run at the same frequency, even on the same chip. Without having to communicate the processes ran much faster, taking only 5ms to 10ms to finish each run. Another interesting feature of the no communication version is that it has much higher variability, since the first component only accounts for around 20% of the data as opposed to around 40% for MPI or sos.

Mutual exclusion should appear as states away from the center line because it prevents processes from being in the same states, but this absence of points is hard to see. When comparing dining philosopher graphs Figure 4,5 we do see a difference in symmetry which we attribute to missing states for the blocked philosopher. In examining the first two principal components of the dining philosophers runs, Figure 11, first note the difference in the amount of variation the graph accounts for - about 70% for the symmetric case, and only about 45% for the asymmetric case. This shows that the asymmetric case has significant variance in higher modes, which can't be seen, and is a strong indicator that something is up. Second note the heavier density of points. Since an asymmetry was forced, more transitions happen, resulting in a greater density of points. Finally, and most tellingly, the variation of the second component in the positive direction is reduced by 1, indicating a reduction in philosophers making it into the eating stage, 8. Note the average state will never be 8, since less than half of the philosophers can make it into the eating state. If the principal component is 5 and the secondary is three, then at least one philosopher is likely eating. If the secondary is greater than three, then likely more than one is eating. For the asymmetric problem it is far less likely for multiple to eat, as indicated by the graph.

Figure 11: First two principal component of dining philosophers with starvation (lower) and without starvation (upper). Each graph shows four independent executions denoted by color (blue and green lines are from the akek cluster at CSUSB while the red and cyan lines are from the computer eris at CSUSB). The percentage is the average percent of the data's variation that is seen in the graph.





6.5 Effects of Network

To see the effects of a Network on an execution, we ran the a parallel prefix code on a single node of the Baylor Tardis cluster (each node has two 12 core Intel Xeon E5-2697v2 cores and the nodes are connected by 40Gb/s Infiniband), and then on three nodes of the same cluster. Each run was composed of five separate prefix calculations done in sequence and four independent runs are shown in Figure 12. The five separate runs are easy to distinguish, as are the stages of the prefix calculation. Each run has an initial flat section between 0 and 1 where the processes are initializing. Once this is completed stage 1 takes place, which is the initial division and node sums, corresponding to the rising section of the graph. Stage 2 computes the prefix sums and corresponds to the slightly decreasing section of the graph. The final rising portion of the graph is the cleanup after. What is particularly interesting is the effect of the network is surprisingly small on this problem. The effect is best seen in the plot of the first two principal components, of Figure 13. The clump on the left is initialization, the arch is stage 1, and the clump on the right is stage 2 and cleanup. The network effects are subtle, but notice first that in the clump on the left, the networked version is more likely to be in block 0 and the single node spends more time relatively in block 1. Since block 0 is setup and involves initial distribution of data, this takes longer on the net and shows up in the graph. The second place a difference can be seen is where the arch (stage 1) connects to the right clump (stage 2). This is where network activity takes place and happens in basic blocks 2, 3, and 4. This area is more dense in the networked version. Both of these effects are subtle, but illustrate that even in extremely similar runs PCA graphs of the phase space highlight the network activity.

6.6 Amoeba Simplex Entropy

We calculate entropy for the four modes of optimization code in table 3.

Entropy calculated values match expectations from figures 1, 2, 3, showing entropy increasing as

Figure 12: Principal component of a 16 process parallel prefix execution versus sample point. Each run has five separate prefix calculations done in sequence and four independent runs are shown by color (blue, green, red, and cyan). The upper graph shows the results of the 16 process run on a single node that had two 12 core Xeon E5-2697v2 processors. The lower graph shows the result of running the 16 processes on 3 nodes with the same chipset over a 40Gb/s Infiniband network. The percentage is the average percent of the data's variation that is seen in the graph.

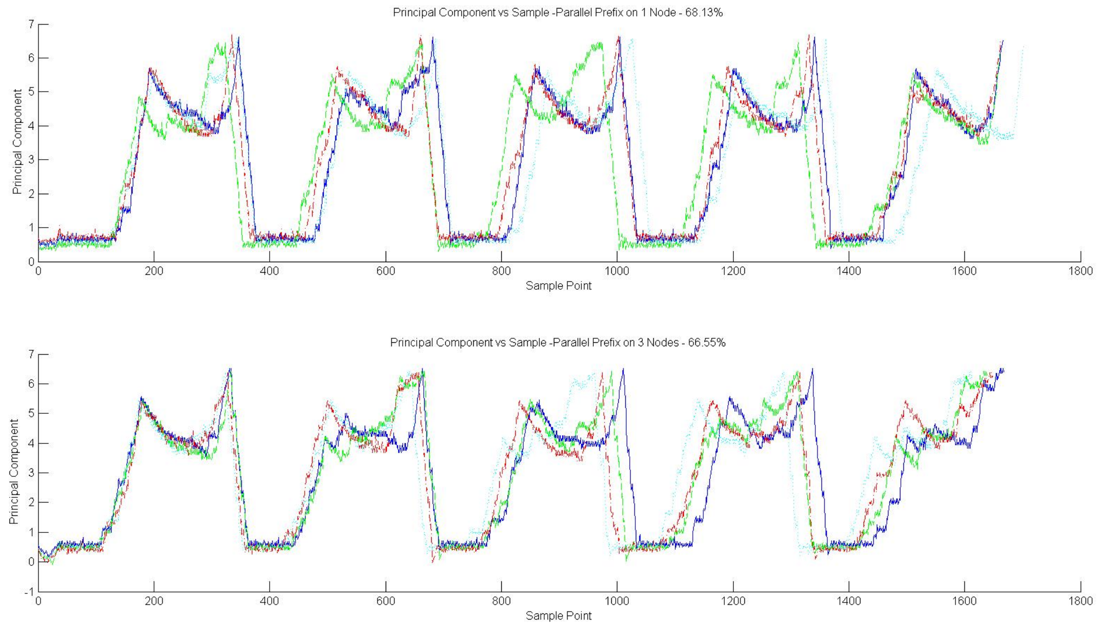


Table 3: Entropy 17D and 32D space

	AKEK-1	AKEK-2	TARDIS-3	TARDIS-4
MPI (synchronous)	5.37	8.94	9611.89	21912.77
SOS (asynchronous)	7.55	17.53	548227.36	391717.29
no communication	20.04	45.40	6833.61	9380.21
no comm., guided	84.05	57.20	869427.34	1579656.30

Data sets aggregate 5 runs of 20 iterations for each communication mode, for same code and starting data. Set 1 ran on 6 networked nodes, set 2 used a single 24 core node on AKEK. Sets 3 and 4 were 32 processes on a single 48 core node on TARDIS.

Figure 13: First two principal component of a 16 process parallel prefix execution. Each run has five separate prefix calculations done in sequence and four independent runs are shown by color (blue, green, red, and cyan). The upper graph shows the results of the 16 process run on a single node that had two 12 core Xeon E5-2697v2 processors. The lower graph shows the result of running the 16 processes on 3 nodes with the same chipset over a 40Gb/s Infiniband network. The percentage is the average percent of the data's variation that is seen in the graph.

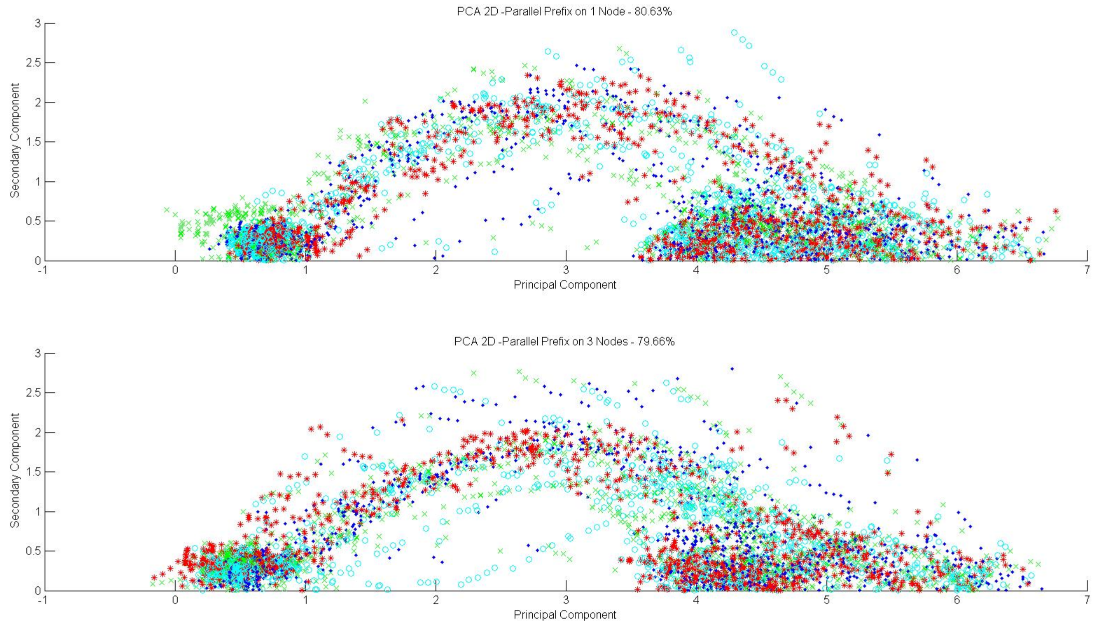


Table 4: Dining philosophers entropy

	AKEK 1	AKEK 2	ERIS 1	ERIS 2
Symmetric	- 76	- 87	- 70	- 70
Live-lock	- 170	- 184	- 190	- 189

All cases were 1000 iterations, showing runs 1 and 2 on two different machines. Live-lock was simulated by blocking philosopher 5 from acquiring 2 forks.

Table 5: Dining philosopher subsets

	AKEK sub5	AKEK sub7	ERIS sub5	ERIS sub7
Symmetric	- 6.34	- 6.20	- 6.74	- 6.82
Live-lock	- 3.07	- 5.37	- 2.94	- 5.91

Sub5 includes processes 4,5,6. Sub7 includes 6,7,8.

synchronization constraints are reduced or absent.

6.7 Dining Philosophers Entropy

We use the combinatorial approach to calculate entropy for dining philosophers. Different code, dimensionality and state definition prevent direct comparison with table 3 .

Table 4 shows a much stronger difference between the symmetric and the live-locked executions than was visible in graphs 4,5. We attribute reduced entropy of the live-lock case to less variability in states, since some legal states are unreachable.

We also calculate the entropy of the states for subsets of three adjacent philosophers in a given run, including and not including the blocked philosopher. From table 5, we see that we can narrow live-lock down to a particular process by computing the entropy of subsets of states, once alerted by data such as table 4.

7 Conclusion

We have shown how to define parallel state, derive a phase space for entropy calculations, and calculate the entropy using combinatorial analysis. A main contribution of this work is the introduction of an experimentally measurable phase space, which we then use to analyze execution states, ensembles of states, and ensembles of complete executions.

The phase state is validated by the extraction of consistent entropy values that are predictable from the program code and execution mode, and the detection of features such as synchronization properties that become visible in the numeric data and in the graphs.

The analysis also reveals unexpected features of execution - specifically the much greater than expected divergence from a coordinated lockstep execution, the ability to identify features such as the position in code where asynchronous communication is required to block and the presence of live-lock and starvation in a subset of processes.

The PCA analysis shows a capability to extract execution details at the level of individual process states, and reveal hardware properties such as network or memory communications. The ability to extract hardware properties from the phase space further validates it. This analysis further allows visualization of how processes progress through the code, and of the presence of task parallelism.

All of the above warrant further work and experimentation, on a greater variety of platforms and with a larger number of sample codes. We suggest that a reduction in entropy of the execution ensemble must require work, based on a correspondence between entropy of computation and entropy in statistical mechanics, and this work may be reflected in execution time or energy use. We have indications in prior work [19], [18] as well as our present results that greater entropy of execution correlates with speedup.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1985.

[2] K. R. Apt and E. Olderog. *Verification of sequential and Concurrent Programs*. Graduate Texts in Computer Science. Springer-Verlag, London, UK, second edition, 1997.

[3] D. L. Arendt, R. Burtner, D. M. Best, N. D. Bos, J. R. Gersh, C. D. Piatko, and C. L. Paul. Ocelot: user-centered design of a decision support visualization for network quarantine. In *Visualization for Cyber Security (VizSec), 2015 IEEE Symposium on*, pages 1–8, Oct 2015.

[4] L. Bai, E. R. Hancock, L. Han, and P. Ren. Graph clustering using graph entropy complexity traces. In *Pattern Recognition (ICPR), 2012 21st International Conference on*, pages 2881–2884, Nov 2012.

[5] L. Bai, E. R. Hancock, P. Ren, and F. Escolano. Directed depth-based complexity traces of hypergraphs from directed line graphs. In *Pattern Recognition (ICPR), 2014 22nd International Conference on*, pages 3874–3879, Aug 2014.

[6] C. Bandt and B. Pompe. Permutation entropy: A natural complexity measure for time series. *Phys Rev Lett*, 88(17):174102, 2002.

[7] Penelope Boston, Jane Curnutt, Ernesto Gomez, Keith Schubert, and Brian Strader. Patterned growth in extreme environments. In *Proceedings of the third IEEE international conference on space mission challenges for information technology, Citeseer*, pages 221–226, 2009.

[8] X. Chen, L. Wang, A. Y. Zomaya, L. Liu, and S. Hu. Cloud computing for vlsi floorplanning considering peak temperature reduction. *IEEE Transactions on Emerging Topics in Computing*, 3(4):534–543, Dec 2015.

[9] K. Coulomb, M. Faverge, J. Jazeix, O. Lagrasse, J. Marcoueille, P. Noisette, A Redondy, and C. Vuchener. Visual trace explorer (vite). Technical report, 2009.

[10] J. Chassin de Kergommeauxa, B. Steinb, and P.E. Bernardc. Pajé, an interactive visualization tool for tuning multi-threaded parallel applications. *Parallel Computing*, 26(10):12531274, 15 August 2000.

[11] E. P. DeBenedictis, M. P. Frank, N. Ganesh, and N. G. Anderson. A path toward ultra-low-energy computing. In *2016 IEEE International Conference on Rebooting Computing (ICRC)*, pages 1–8, Oct 2016.

[12] Damien Dosimont, Robin Lamarche-Perrin, Lucas Mello Schnorr, Guillaume Huard, and Jean-Marc Vincent. A spatiotemporal data aggregation technique for performance analysis of large-scale execution traces. In *IEEE Cluster 2014*, Madrid, Spain, Sep 2014.

[13] S. Draghici. Using information entropy bounds to design vlsi friendly neural networks. In *Neural Networks Proceedings, 1998. IEEE World Congress on Computational Intelligence. The 1998 IEEE International Joint Conference on*, volume 1, pages 547–552 vol.1, May 1998.

[14] A. T. Freitas and A. L. Oliveira. Circuit partitioning techniques for power estimation using the full set of input correlations. In *Electronics, Circuits and Systems, 2001. ICECS 2001. The 8th IEEE International Conference on*, volume 2, pages 903–907 vol.2, 2001.

[15] Joshua Garland, Ryan James, and Elizabeth Bradley. Determinism, complexity, and predictability in computer performance. Technical report, arXiv preprint arXiv:1305.5408, 2013.

[16] Joshua Garland, Ryan James, and Elizabeth Bradley. Model-free quantification time-series predictability. Technical Report 2014-05-014, Santa Fe Institute, 2014.

[17] E. Gomez. *Single Program Task Parallelism.* PhD thesis, University of Chicago, Chicago, IL, USA, 2004.

[18] E. Gomez and L. R. Scott. Overlapping and shortcutting techniques in loosely synchronous irregular problems. *Springer Lecture Notes in Computer Science*, 1457:116–127, 1998.

[19] E. Gomez and L. R. Scott. Compiler support for implicit process sets. Technical Report TR-2005-14, University of Chicago, Chicago, IL, USA, 2005.

[20] Ernesto Gomez, Keith E. Schubert, and Ritchie Cai. A model for entropy of parallel execution. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 555 – 560. IEEE Conference Publications, 2016.

[21] R.C. Gonzalez, R.E. Woods, and S.L. Eddins. *Digital Image Processing Using MATLAB.* Prentice Hall, New Jersey, 2003. Chapter 11.

[22] A. Hamou-Lhadj. Measuring the complexity of traces using shannon entropy. In *Information Technology: New Generations, 2008. ITNG 2008. Fifth International Conference on*, pages 489–494, April 2008.

[23] Lin Han, Francisco Escolano, Edwin R. Hancock, and Richard C. Wilson. Graph characterizations from von neumann entropy. *Pattern Recogn. Lett.*, 33(15):1958–1967, November 2012.

[24] M. T. Heath and J. A. Etheridge. Visualizing the performance of parallel programs. *IEEE Software*, 8(5):29–39, Sept 1991.

[25] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice Hall International, 1985.

[26] C. J. Honey, R. Kötter, M. Breakspear, and O. Sporns. Network structure of cerebral cortex shapes functional connectivity on multiple time scales. *Proceedings of the National Academy of Sciences*, 104(24):10 24010 245, 2007.

[27] Katherine E. Isaacs, Alfredo Gimnez, Ilir Jusufi, Todd Gamblin, Abhinav Bhatele, Martin Schulz, Bernd Hamann, and Peer-Timo Bremer. State of the art of performance visualization. In R. Borgo, R. Maciejewski, and I. Viola, editors, *Eurographics Conference on Visualization (EuroVis) (2014)*, STAR State of The Art Report, 2014.

[28] R. K. Jana, G. L. Snider, and D. Jena. Resonant clocking circuits for reversible computation. In *Nanotechnology (IEEE-NANO), 2012 12th IEEE Conference on*, pages 1–6, Aug 2012.

[29] R. K. Jana, G. L. Snider, and D. Jena. Energy-efficient clocking based on resonant switching for low-power computation. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 61(5):1400–1408, May 2014.

[30] H. F. Jordan. *The Force.* MIT Press, 1987.

[31] Laxmikant V. Kalé, Gengbin Zheng, Chee Wai Lee, and Sameer Kumar. Scaling applications to massively parallel machines using projections performance analysis tool. *Future Generation Computer Systems*, 22(3):347–358, 2006.

[32] Jinpyo Kim, Sreekumar V. Kodakara, Wei-Chung Hsu, David J. Lilja, and Pen-Chung Yew. *Dynamic Code Region (DCR) Based Program Phase Tracking and Prediction for Dynamic Optimizations*, pages 203–217. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.

[33] Andreas Knüpfer, Holger Brunst, Jens Doleschal, Matthias Jurenz, Matthias Lieber, Holger Mickler, Matthias S. Müller, and Wolfgang E. Nagel. *The Vampir Performance Analysis Tool-Set*, pages 139–155. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[34] Babak Bagheri L. Ridgway Scott, Terry Clark. *"Scientific Parallel Computing"*. Princeton, 2005.

[35] R. Lamarche-Perrin, L. M. Schnorr, J.-M. Vincent, and Y. Demazeau. Evaluating trace aggregation through entropy measures for optimal performance visualization of large distributed systems. Research Report RR-LIG-037, Laboratoire d'Informatique de Grenoble, France, 2012.

[36] X. Li, G. Ouyang, and D. A. Richards. Predictability analysis of absence seizures with permutation entropy. *Epilepsy research*, 77(1):70–74, 2007.

[37] E. Macii and M. Poncino. Exact computation of the entropy of a logic circuit. In *VLSI, 1996. Proceedings., Sixth Great Lakes Symposium on*, pages 162–167, Mar 1996.

[38] S. Mamataj and B. Das. Efficient designing approach of different synchronous cyclic code counters by sequential circuit elements of a novel reversible gate. In *Computational Intelligence and Communication Networks (CICN), 2014 International Conference on*, pages 1031–1036, Nov 2014.

[39] Jon Mathews and R. L. Walker. *Mathematical Methods of Physics Second Edition*. W. A. Benjamin, 1970.

[40] K. Mohror and K.L. Karavanic. Evaluating similarity-based trace reduction techniques for scalable performance analysis. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, SC 09*, page 55:155:12, New York, NY, USA, 2009. ACM.

[41] T. Myktowicz, A. Diwan, and E. Bradley. Computers are dynamical systems. *Chaos*, 19:033124, 2009.

[42] G. Pagano, D. Dosimont, G. Huard, V. Marangozova-Martin, and J. M. Vincent. Trace management and analysis for embedded systems. In *Embedded Multicore Socs (MCSoC), 2013 IEEE 7th International Symposium on*, pages 119–122, Sept 2013.

[43] Roger Penrose. *The Road To Reality*. Knopf, 2005.

[44] Y. B. Pesin. Characteristic lyapunov exponents and smooth ergodic theory. *Russian Mathematical Surveys*, 32(4):55, 1977.

[45] X. Ren, Qihang Yu, B. Chen, N. Zheng, and P. Ren. A real-time permutation entropy computation for eeg signals. In *The 20th Asia and South Pacific Design Automation Conference*, pages 20–21, Jan 2015.

[46] J. Riihijärvi, M. Wellens, and P. Mähönen. Measuring complexity and predictability in networks with multiscale entropy analysis. In *INFOCOM 2009, IEEE*, pages 1107–1115, April 2009.

[47] Draghici S. and V. Beiu. Entropy based comparison of neural networks for classification. In *Proc. of The 9-th Italian Workshop on Neural Nets, WIRN Vietri-sul-mare*. Springer-Verlag, 22-24 May 1997.

[48] Lucas M. Schnorr and Arnaud Legrand. *Visualizing More Performance Data Than What Fits on Your Screen*, pages 149–162. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[49] Carmen Sigovan, Chris W. Muelder, and Kwan-Liu Ma. Visualizing large-scale parallel communication traces using a particle animation technique. In B. Preim, P. Rheingans, and H. Theisel, editors, *Eurographics Conference on Visualization (EuroVis) 2013*, volume 32, 2013.

[50] Brian Strader, Keith E Schubert, Ernesto Gomez, Jane Curnutt, and Penelope Boston. Simulating spatial partial differential equations with cellular automata. In *BIOCOMP*, pages 503–509, 2009.

[51] J. A. Tierno, R. Manohar, and A. J. Martin. The energy and entropy of vlsi computations. In *Advanced Research in Asynchronous Circuits and Systems, 1996. Proceedings., Second International Symposium on*, pages 188–196, Mar 1996.

[52] L. R. Varshney. Toward limits of constructing reliable memories from unreliable components. In *Information Theory Workshop - Fall (ITW), 2015 IEEE*, pages 114–118, Oct 2015.

[53] J. De Weerdt, S. vanden Broucke, J. Vanthienen, and B. Baesens. Active trace clustering for improved process discovery. *IEEE Transactions on Knowledge and Data Engineering*, 25(12):2708–2720, Dec 2013.

[54] Shu-Ming Armida Yang, Chuen-Tsai Sun, and Ching-Hung Hsu. Energy, matter, and entropy in evolutionary computation. In *Evolutionary Computation, 1996., Proceedings of IEEE International Conference on*, pages 196–200, May 1996.

[55] L. Yu-hang, Z. Ming-fa, W. Jue, X. Li-min, and G. Tao. Xtorus: An extended torus topology for on-chip massive data communication. In *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2012 IEEE 26th International*, pages 2061–2068, May 2012.

[56] X. Zhao, Y. Guo, X. Chen, Z. Feng, and S. Hu. Hierarchical cross-entropy optimization for fast on-chip decap budgeting. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 30(11):1610–1620, Nov 2011.