NOTES ON VISUALIZATION OF PARALLEL EXECUTION

ERNESTO GOMEZ, KEITH SCHUBERT, RITCHIE CAI

We have long known that parallel execution is not deterministic - one of the issues of parallel programming is ensuring determinism through synchronization; which adds complication to programming and introduces the possibility of deadlock.

Our parallel state and transition notation from Section 1 (introduced in E. Gomez dissertation) implies non-determinism. We knew at the time that the states we proposed were real because they were measurable (in principle at least), and that tying them to basic blocks allowed us to analyze the execution in terms of events in the code, rather than the coarser approach of looking at interactions between processes. Further, the notation implies the existence of a P-dimensional phase space (for P processes) for parallel execution, in which each point is a P-tuple of the basic block numbers in each state. In such a space, execution would be represented as a path in a P-dimensional hypercube from the start state to the end state. Furthermore, the section of this phase space inhabited by an ensemble of repeated executions of the same code should help us investigate the properties of the execution.

Gomez gave a seminar at CSUSB in 2002 using a simulation of a two-process execution, and showing a parametric graph of the basic blocks for each process as a function of time. Again, no attempt was made to actually instrument a real execution, and there was no idea of representing execution in terms of phase space (notes from seminar are at http://www.cse.csusb.edu/egomez/ following the link: Synchronization and the Modeling of Parallel Execution).

An additional motivation for this work is a long-standing idea that synchronization increases order in parallel execution, and that this increase in order has to come at some cost over and above the direct cost of synchronizing. This concept is touched on in "Overlapping and Shortcutting Techniques in Loosely Synchronous and Irregular Problems". Ernesto Gomez and L. Ridgway Scott, LNCS 1457, Springer, August 1998 which makes the point that there are synchronization wait costs over and above the direct communication cost of synchronizing. Our prior simulation of a theoretical 2 process execution supports this idea, but to really investigate requires instrumenting real parallel executions and studying their properties.

We (Keith Schubert (now at Baylor) and Gomez) have speculated about the possibility of applying the concepts of entropy and work to parallel execution; the state space and visualization techniques discussed here lead into and support this speculation.

We here develop a practical method for instrumenting parallel execution to extract parallel state as a function of time, and to generate the path of an execution in phase space. We develop a method for displaying P and P+1 dimensional paths and sets of states in two and three dimensions which we believe gives a reasonable intuitive grasp of the properties of the P-dimensional original. We apply the methods described to a particular parallel algorithm, which we implemented using different types of communication as well as no communication. In our conclusions we develop some preliminary findings and suggestions for future investigation.
NOTES ON VISUALIZATION OF PARALLEL EXECUTION

1. Assumptions

We consider a parallel execution in the static SPMD model - we have a set \( \Gamma \) of \( P \) processes, all of which start in a basic block labeled \( s \) and end in a basic block labeled \( e \), and each of which follows a path \( p_i = s \rightarrow^* e \) through a control flow graph \( G = (V,A,s,e) \) (where \( V \) is a set of nodes denoting basic blocks in the code, \( V \) is a set of directed arcs between nodes, \( i \in \Gamma \) is process number).

We consider a transition from one basic block to another (following an arc in \( A \)) to be a step in execution, numbered in sequence as the process follows path \( p_i \). We denote the state of process \( i \) by \( \sigma_{i,j} = (x_j,M_j) \) where \( j \) denotes the step in execution, \( x \) the block being executed, and \( M \) the contents of memory at that process. Knowing \( M \) and \( x \) we may determine the next step process \( i \) will take and the contents of memory at the start of the next block. By induction we can determine the rest of the execution of \( i \) from \( \sigma_{i,j} \) (if we do not introduce random factors or external data).

We define the state of a parallel execution as the set of the individual process states that exist concurrently (at a given external time):

\[
S_{\Gamma,J} = \{ \sigma_{i,j} \}
\]

This is a total state of all processes in \( \Gamma \), replacing \( \Gamma \) with \( I \subset \Gamma \) gives \( S_{I,J} \) which is a partial state involving a subset of processes. We can then describe a parallel execution in total states \( S_{\Gamma,s} \rightarrow^* S_{\Gamma,e} \) - and note that an execution in partial states must be a DAG such that any slice that partitions the DAG by cutting concurrent paths must be consistent with a state in the total execution.

The restricted execution model we discuss here is for initial convenience in developing the experiments, and does not preclude extending these concepts to a more general model.

2. State

2.1. Instrumenting the code. It is not practical to save \( M_j \) for each process at each step - we use the basic block \( x_j \) as our state proxy and the path \( p_i = s \rightarrow^* e \) as a proxy for process execution. 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.

In the code, we number basic blocks sequentially from \( s \) to \( e \). We define an \textit{out} \((b)\) command, where \( b \) is block number; \textit{out} prints the triple “\( t \) \( b \) \( pid \) ” to file “\text{name+pid.out}”, creating one .out file per process. Here “\text{name}” is the program name, “\text{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 “\text{name+ident.srt}” (where \text{ident} is a label for a specific run), 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. Block identification and instrumentation is presently done by hand, so is only approximately related to actual basic blocks. It is recognized that 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 which is consistent with states in \( S_{\Gamma,s} \rightarrow^* S_{\Gamma,e} \), though possibly missing some detail.
2.2. **Reconstructing parallel state.** For a $P$ process execution we represent each state as an ordered $P+1$ tuple, where the first $P$ places are the basic block executed by the process with pid corresponding to position in the tuple, and the $P+1$ place holds the time. For example "0 0 1 2 50" would be a state of a 4 process execution, in which processes 0 and 1 are each in block 0, process is in block 1 and process 2 is in block 2, 50 microseconds after start. The execution is placed in a $M*(P+1)$ matrix, where each row represents a state, and there is one row for each of the $M$ total states identified in the execution.

To construct the states from the .srt file, we use the following algorithm:

```
Pseudocode based on Scilab code:

```matlab
mx = 1 // raw_data 1: time
    // 2: block
    // 3: pid (from 0 to P−1, must increment for
    //    Scilab index convention)

// initialization

int resolution=10 // microseconds
int N // number of rows in raw_data .srt file
int raw_data(N,3) // read .srt file into matrix
int states(P+1,M) // M not known initially, but of order N/P −
    // allocate extra and truncate
    // data matrix initialized to 0
int siz=P+1 // P is number of processes
int row=1 // start value

// loop reads raw data and creates states matrix

for i=1:N
    if abs(raw_data(i,1)−states(row,siz))<=resolution then
        states(row,raw_data(i,3)+1)=raw_data(i,2);
    else // time has changed
        row=row+1; // new state row number
        states(row,siz)=raw_data(i,1); // set new time
        for j=1:siz−1 // copy row, except time
            states(row,j)=data(row−1,j);
        end // copy row loop
        states(row,raw_data(i,3)+1)=raw_data(i,2);
    end // time has changed
end // read loop

M=row; // number of rows in states matrix

// writes states file "data+fid.name"
```
We assume start time=0. If two entries in raw data have the same pid and different times, the data from the earlier time will be overwritten - this is not a frequent issue in current data, but with faster systems or to capture detail on small basic blocks we will have to increase time resolution. If a process does not have an entry to match a particular time, the block from the previous state will be copied. This captures the possibility that some blocks will take significantly more time to execute than others, so it is possible for a particular process to take consecutive steps making new parallel states, while other processes in the same state do not advance. This possibility is particularly relevant for partial state execution, in which concurrent process subsets are executing different tasks.

2.3. **Phase space.** Our definition of state as a P-tuple of basic block numbers leads to a phase space of all possible states as a P-dimensional hypercube of side $e - s$ (where $e$ is the number of the end state, and $s$ is the number of the start state). The state (block value) of any particular process is recorded as the coordinate value for dimension $j$, where $j$ is the process number; the tuple $P$ of all the process block values is a point in (inside or on the boundary) of the hypercube.

Our base case is $P$ processes without interaction, each therefore independent of the other processes. Each process can be in set of states denoted by integers $0 \ldots e$, where $0$ is taken as the number of start state $s$ and $e$ is the number of the end state.

Each processes can transition independently from all others, so processes can be considered orthogonal. Therefore we represent a phase space for a parallel execution as the hyper-volume of a P-dimensional hypercube, with sides of length $e$. A point inside (or on the boundary) of the hyper-volume is a P-tuple; if we restrict the elements of the tuple to integers, we get all possible states.

States of execution with interaction between processes are certainly a subset of the states of non-interacting executions, so the same phase space works. In either case an execution may be represented as a polyline path in the hypercube, constrained by the requirement that each vertex of the polyline must be described as a P-tuple of integers. 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.

We note that the states $P$ represented in the phase space are not in fact the complete parallel state $S_{\Gamma,j}$. In particular, we have not found a practical way of representing contents of memory, and the phase space represents only total states of processes in $\Gamma$, but does not represent partial states of $G \subset \Gamma$; furthermore it needs to be extended to allow the possibility that an initial set of $P$ processes may create or destroy processes during an execution.

We further note that the phase space allows only states with integer values of all coordinates; and that the control flow graph of the code restricts the possible successor states for any given point $P$. An implication of the control flow graph is that the successor state for a given process $p$ represented in a state $P$ does not have to be adjacent to $P$ in the phase space, since it might represent a jump to a block of code that is not the immediate successor of the block $P$ is in.

2.4. **Evolution in time.** The path in phase space is a parametric graph of the progress of individual processes, with time as the parameter.

We may include time explicitly in the representation by adding a dimension $t$ orthogonal to the hypercube, in which case the path would advance in time at each step and
would have no cycles since the time coordinate is monotonically increasing. In this case we would have a polyline transiting through a set of phase-space cubes labeled \( t_0 \) to \( t_e \), starting at \((0,0,\ldots,0)\) on the first cube, and ending at \((e,e,\ldots,e)\) on the last cube.

2.5. **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 from the center. For better visualization of multidimensional data, we use a 3D perspective graph displayed on a plane - we use Scilab for display, so we can rotate the 3D 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. Denote the horizontal plane with the letter \( R \), then the projections of the 3 axes \( x,y,z \) are separated by angles of \( \frac{2\pi}{3} \); for convenience we display \( x \) at an angle of 0.

For processes - dimensions \( P > 3 \), we follow the same scheme, but now separating the axes by \( \frac{2\pi}{P} \). We then calculate the \((x,y,z)\) coordinates for unit vectors \( u_i \) vectors lying along each of the dimensions we wish to display. The point \( X_p = (x_1,\ldots,x_p) \) in \( P \) dimensions is placed in 3D at: \( X_3 = \sum_{i=1}^{P} (u_i x_i) \) 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 of the phase space 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).

To also graph time, we flatten the projection onto the XY plane by dropping the z coordinate of \( X_3 \) and replacing it with \( t \). The visualization space now looks like a vertical cylinder centered on the origin.

This scheme has the advantage of being fairly intuitive. Specifically, if processes advance in lockstep through the same code, then every state should look like \((k,k,\ldots,k)\) where \( k \) is a constant 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. Excursions away from the center line should reflect processes advancing at different rates or subsets of processes doing different tasks concurrently.

It has the disadvantage of introducing an artificial sense of proximity between consecutively numbered dimensions - for example \((2,0,0,2,0,0)\) would be plotted on the vertical, whereas in 6D it would be as far from the vertical as \((2,2,0,0,0,0)\)- which would appear at a large angle from vertical. The 3D plot makes some points appear closer to the diagonal than they really are; it further distorts the distances from the origin, making them appear larger because the unit vectors used for display are not orthogonal.

No attempt has been made to compensate for distortion, so scales on phase space 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.

We will also use 2D graphs which are projections of our 3D graphs onto the XY plane (view from the top) or onto the XZ plane (side view).

2.6. **Ensemble of executions.** Parallel transition \( S_{\Gamma,I} \rightarrow S_{\Gamma,J} \) is non-deterministic, because absent synchronization any combination of processes in \( \Gamma \) may transition, giving
a new state. As a result, each individual parallel execution may follow a different path through phase space even for the same code, number of processes, data and machine.

Therefore to be able to study the behavior of parallel code, we need an ensemble of executions of the same code, number of processes, machine and data to allow us to distinguish features intrinsic to the algorithm and code from features that may vary randomly from one execution to another.

3. Experiments

3.1. Test code and system. Baseline code was an implementation in PC of the modified amoeba downhill simplex algorithm from E. Gomez dissertation. 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.

Four variants of the code were used - labeled “none”, “answer”, “sos” and “mpi”.

The SOS code was written in the most recent (2.93 - experimental) version of PC/sos, and implements non-blocking, out of order send, receive and collective communication. The SOS code was used as base for the other versions. Code is a modified downhill simplex algorithm for N-dimensional minimization, using N+1 processes assigned to each vertex of the simplex.

The MPI code replaced a single SOS broadcast at the end of each iteration with the equivalent MPI code. Mpi broadcasts are blocking code at all receiving processes, and use a fixed communication tree.

The “none” code had all communication statements remarked out, and was hard coded to stop in 20 iterations (this version did not converge because processes did not have access to data from other processes).

The “answer” code did not communicate, but had a matrix with the data computed by the SOS code at each iteration; the “answer” code therefore used the same partially computed data of the SOS iteration and converged on the same computational path.

Code was instrumented with “out” statements (see 2.1) which saved time, block number and process number. 11 locations in the program were marked from 1...11, giving a hypercube of side 11. Locations 2-10 were inside the minimization loop.

Each of the 4 variants was run 5 times for a 16 dimensional (17 process) problem; SOS, MPI and none were also run on a 6 dimensional (7 process) problem. 16 dimension runs were on AKEK, a parallel research cluster with 7 nodes of 24 Intel Xeon 2.4 GHz cores each, communicating over gigabit Ethernet, with SOS and mpich 2.1.3. The 7 dimensional runs were on ERIS, a 4 core (with hyperthreading) Intel i5 2.5 GHz machine, using SOS and mpich 3.1.2 and the ch3:tcp communication system. (In each case multiple sets of data were collected during code development, with results consistent with those reported here, which were generated with the latest versions of the code).

3.2. Time display. Single process, PC/sos, 17D+time. Close examination of the graph in a tool allowing rotation and zoom confirms an acyclic polyline.
The ensemble graph does not include the path, because the lines obscured the points. Although not connected, the distribution of points makes it evident that each of the 5 runs took a different although qualitatively similar path. The details of interest appear
3.3. **17 Dimension phase space.** 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, 2D are XY or XZ with X horizontal. All runs on AKEK, which has enough cores to assign one per process.

Scales represent the 3D projection of 17D euclidean distance from the origin - given that the program starts in block 1 and ends in block 11, these distances are roughly indicative of where a given state is in the code, but are useful mainly for comparison between different graphs.
No communications 17d.
No communication answer matrix
NOTES ON VISUALIZATION OF PARALLEL EXECUTION

MPI 17d
No comm. XY plane 17d
Answer matrix, XY 17d
PC/sos XY plane 17d
MPI XY plane 17d
Overlay answer+sos+mpi XY 17d
no comm. XZ 17d
answer matrix XZ 17d
NOTES ON VISUALIZATION OF PARALLEL EXECUTION

PC/sos XZ 17d
overlay answer+sos+mpi XZ 17d
3.4. 7D phase space. Each of the graphs displays results for 5 runs of 20 loop iterations and 7 concurrent processes for the amoeba program. 3D graphs have standard XYZ coordinates, 2D are XY or XZ with X horizontal. All runs on ERIS, a 4 core machine; only one of the cores could be dedicated to a single process while the other 3 cores would each get 2 processes. Scales on graphs are 3D projections of 7D euclidean distance of state from the origin.
no comm. 7d
NOTES ON VISUALIZATION OF PARALLEL EXECUTION

PC/sos 7d
no comm XY 7d
PC/sos XY 7d
MPI XY 7d
no comm. XZ 7d
PC/sos XZ 7d
MPI XZ 7d
3.5. **Observations on experimental method.** Instrumenting the basic blocks and reconstructing parallel state outside the actual execution provides a minimally invasive way of seeing what is happening in a real execution.

A first observation is that the phase space representation as seen in 3.3 and 3.4 is more useful than the execution over time displayed in 3.2, because the phase space directly relates to the code in a way that the time representation does not, and shows more detail.

A second observation is that our expectation that each execution is different and that the ensemble of multiple executions is the right object of study is justified, if what
we want to study is general properties of parallelism. A study of a single execution may still be of interest to observe what happened in a particular case, but it would not tell us much about future executions of the same code.

The direct correspondence between our notation for parallel state and execution (2) and a straightforward and informative way of modeling actual execution speaks to the usefulness of our notation.

4. **Preliminary conclusions and surprises**

4.1. **Non-determinism.** All executions were on lightly loaded systems which were only running operating system functionality in addition to our test code. Particularly in section 3.3 documenting the 17 process runs, the system was able to assign one core to each process, and all cores were the same type and speed - it was an essentially homogeneous system. We expected some variation between different executions. We were surprised by how much variation we saw.

Specifically - in the 3D graphs, and in the graphs showing projection on the XZ plane - a lockstep execution in which all processes transition together would be represented by a vertical line from the origin. We expected some excursions from that line, but we also expected that the general trend would leave most states close to the center line. The results totally contradicted this expectation.

4.2. **Influence of data on phase space.** Comparing the XY 3.33.3 and XZ 3.33.3 projections of “none” and “answer” code (both without communications) significant differences are visible. The “none” code was not converging, and its states appear more spread out from top to bottom on the XZ projection. The “answer” code, which followed a path to convergence, was executing more code in higher-numbered blocks. This makes sense in that the lower numbered blocks implement movement of the test simplex toward a minimum, and higher numbered blocks implement the part of the algorithm where the simplex shrinks around a minimum.

The “answer” code also displayed significantly more excursions away from the center line of the phase space, similar in this case to the “SOS” code and more than the ‘mpi’ code.

4.3. **No tendency toward the center.** We expected that, after an excursion away from the center line, there would be a tendency to go back. That is, if process p runs faster than the others and transitions first - producing an off-center state, then other processes would be likely (on a homogeneous system) to transition before p steps again, moving the next state back.

Instead, we find that there is little if any indication of a trend to stay near the center. 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 - see section 2.5.

The distribution of next states seems more akin to a random walk than anything else. Particularly the ‘SOS’, and both no communication executions seem to expand into all areas of the phase space.

4.4. **Clumping unrelated to synchronization.** We expected based on earlier modeling that synchronization or communication would introduce clumping in the phase space. We do see that particularly in the MPI execution.

We had not expected the horizontal banding visible in the XZ projections. However, in retrospect we should have. Since we only instrumented 11 blocks in the code, and
since all state coordinates are integer, we should have expected that states would line up horizontally.

We however did not expect that states would clump radially out from the center line. This is particularly easy to see in the "none" executions 3.4,3.4,3.4. It is also visible to a lesser extent in the 17d graphs 3.3. The effect seems more surprising in the code with no synchronization. Part of the effect may also be due to the integer state space, but the separation between clumps does not seem to have the right scale.

4.5. Effects of synchronization. Synchronization is carried out by making processes wait. This should be particularly true for barriers, which are the most restrictive synchronization. Our test code has two global collective communications - an allreduce followed by a broadcast. Both these communications require every process to wait for data, so they act like barriers. Both communications appear near the end of the loop, in blocks 8 and 9, so we expect clumping near the top.

The MPI graphs show very strong clumping - processes reach the last blocks of the loop and wait there for others so we see many marked states near the top of the MPI 3D graphs and the MPI XZ projections.

SOS uses overlap communications, so communication statements are all non-blocking. 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 see the effects in the SOS 3D and XZ plane graphs. We have less clumping at the top than for MPI. We also have some clumping at the bottom of the SOS graphs - this is because 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 would show up at the bottom where the iteration starts.

4.6. Relation of phase space to execution of specific code. At the end of a loop iteration, there is an allreduce that determines which process is closest to convergence. This is followed by a broadcast from the process with the best values. The root of the allreduce defaults to process 0, but the root of the broadcast is some different process. Looking at the projections onto the XY plane for SOS and MPI in sections 3.3 and 3.4 shows a clump near the center, possibly corresponding to the reduction, and a clump off-center which perhaps is the broadcast. This needs confirmation - it is otherwise a striking coincidence which is not visible in the codes with NO communications.

There are visible differences in the phase space occupied by two codes without communication, but which exercise different routines (since the "answer" code converges and the "no" code does not).

4.7. Usefulness of phase space. This very preliminary investigation both supports prior theory and reveals surprises about the nature of parallel execution. It affirms the usefulness of the state representation and supports the desirability of continuing experiments and theoretical work.

4.8. Order in the phase space. Order in a phase space may be defined in terms of what fraction of all possible states are occupied by the system. It seems evident that the MPI code is far and away the most ordered. The SOS and the "answer" code with no communication have similar spread through the phase space (both show significantly more spread than the "no" communication code in section 3.3). However, SOS does show more clumping/order at the top and bottom of the space than the "no"
communication code, so we will still claim that the SOS execution is more ordered, although significantly less ordered than MPI.

We propose the ordering is related to the amount of work needed to synchronize. The "no" communication code is the least ordered, and does no work to increase order. The MPI code is the most ordered and implements the most restrictive synchronization which should require the most work. The SOS code is intermediate in both work and order. The "answer" code appears to be about as ordered as the SOS code, but shows no sign of clumping at low-numbered states.

These observations support a concept of entropy of parallel execution tied to synchronization, and that the work of enforcing stronger synchronization reduces this entropy, and is the subject of continuing work.