VISUALIZING PARALLEL EXECUTION WITH SYNCHRONIZATION

ERNESTO GOMEZ

Note. Based on a talk given 1 May 2002 at the Department of Computer Science, California State University San Bernardino

Visualization of parallel execution may not by itself prove anything. However, it can show features and limitations of particular models and serve as inspiration for the development of theory and practice. We here present a visualization that we have found useful, and show some of its consequences.

All visualization implies some specific model. Ours is the asynchronous execution of multiple copies of the same code, termed SPMD. This model represents common practice in real parallel and distributed computation (for example, the Message Passing Interface - MPI Standard assumes SPMD).

We highlight synchronization as a key feature of parallel execution. In fully asynchronous execution it is possible that some theoretically concurrent process completes its execution before some other process even gets out of a start state. Synchronization of some form is required to simply ensure that interprocess communication happens, as well as to ensure deterministic execution. In addition, synchronous versus asynchronous execution is one of the key parameters used to distinguish theoretical models of parallelism.

1. Definitions:

We will need some definitions and notation.

1.1. **SPMD parallel execution:** We restrict our considerations to *Single Program Multiple Data* (SPMD) execution, with a fixed number of processes:

An SPMD execution is a fixed set Γ of N processes executing the same code, each process has a distinct process i.d., may have different data in memory, and may execute different routines within the same program (also termed *procedural parallelism*).

1.2. **Serial state:** Since SPMD is essentially a collection of serial processes, we need to define serial state:

$\sigma_j = (x_j, D_j)$

denotes a single process executing CFG node x at step j in execution, and has access to data D at that step.

Date: 18 May 2002.

1.3. **SPMD parallel state :** A state of the parallel execution is a collection of the serial states of a group of processes. In some cases we will only be interested in the state of a subgroup $G \subseteq \Gamma$ of the total set Γ of processes.

$S_{G,J} = (x_j^p, D_j^p)$

Here $p \in G$ is a process executing CFG node x at step j in its execution, D_j^p is the data in memory at process p and step j, $G \subseteq \Gamma$ is a group of processes and J is the multi-index of step numbers in which processes in G find themselves.

1.4. **Parallel transition:** Our basic model is that of separate processes running on independent processors; this implies that processes are asynchronous. One consequence is that we don't know which process (or processes) is going to transition to a different serial state at any given time.

A change of parallel state occurs when at least one process, but possibly more than one, changes its serial state. On repeated execution of the same initial state, different processes may advance, so the parallel transition is not deterministic. We have:

$S_{G,J} \to S_{H,K}$ such that $H \cap G \neq \emptyset$,

For at least one $p \in H \cap G$, the index $j_p = k_p - 1$ where $j_p \in J$ and $k_p \in K$ are, respectively, the step number of p in J and in K (that is,a t least one process advances), and $S_{G,J} \to S_{H,K}$ is consistent with some pair of total states such that $S_{\Gamma,J} \to S_{\Gamma,K}$.

1.5. **Parallel execution:** In SPMD we have a single CFG node where all processes start, and at least one (but possibly several) node which is reached at all processes at the end. We presume that the parallel execution completes if every serial execution in it does.

We can define a start state $S_{\Gamma,0}$ where all processes are in the start node, and an end state $S_{\Gamma,End}$ where all processes have completed their execution. If every SPMD process reaches an end state, we have a total execution, in which every transition is between total states:

$E_\Gamma = S_{\Gamma,0} \to^* S_{\Gamma,End}$

In general, the only requirement to assure that such an execution exists is that no individual process fails or diverges, and that interaction between processes does not deadlock.

1.6. **Deterministic parallel transition:** A parallel transition in which all processes advance. This is characteristic of lockstep synchronous execution.

$S_{G,J} \Rightarrow S_{H,K}$ such that $H \cap G \neq \emptyset$.

For every $p \in H \cap G$, the index $j_p = k_p - 1$ where $j_p \in J$ and $k_p \in K$ are, respectively, the step number of p in J and in K (that is,a t least one process advances), and $S_{G,J} \Rightarrow S_{H,K}$ is consistent with some pair of total states such that $S_{\Gamma,J} \to S_{\Gamma,K}$.

If an execution can be represented by transitions of this type, it can be shown to be deterministic. We may be able to show that an execution E with non-deterministic $\to$ transitions is equivalent to an execution D with $\Rightarrow$ transitions, if we can show that D is always part of the ensemble of E executions and that all communications in E occur in states that are the same as those of D.

Note, however, that that for a total execution: $D_\Gamma = S_{\Gamma,0} \Rightarrow^* S_{\Gamma,End}$ to exist with deterministic parallel transitions, it is necessary that all processes have the same number of serial transitions from start to end. Therefore either all processes

are following the same execution path, or some processes are forced to take steps without doing anything.

1.7. **Pstream execution:** A Pstream is a sequence of parallel with a constant group $G \subseteq \Gamma$ connected by deterministic $\Rightarrow$ transitions;

$\Phi_G = S_{G,J} \Rightarrow^* S_{G,K}$

We introduce two operations on Pstreams; a split into disjoint Pstreams:

$Split(\Phi_G) = \{\Phi_{G_1}, \ldots, \Phi_{G_N} | \cup_N G_i = G, G_i \cap G_j = \emptyset \forall i \neq j\}$

and a merge:

$Merge\{\Phi_{G_1}, \ldots, \Phi_{G_N} | \cup_N G_i = G, G_i \cap G_j = \emptyset \forall i \neq j\} = \Phi_G$

Given a CFG with a single end node (we can always transform any CFG to single end node), and an initial Pstream:

$\Phi_\Gamma = S_{\Gamma,0} \Rightarrow^* S_{\Gamma,K}$

we can show the existence of an SPMD Pstream execution:

$E_\Phi = S_{\Gamma,0} \Rightarrow^* Split \Rightarrow^* S_{G,J}/S_{H,K} \Rightarrow^* Merge \Rightarrow^* S_{\Gamma,End}$

Splits occur at CFG nodes of outdegree > 1 in which processes may take different CFG paths, and merges occur in nodes of indegree > 1 in which processes must again follow the same CFG path on exit from the node. Since processes in the same Pstream follow the same CFG path, they take the same number of steps along that path and so their execution can be represented by $\Rightarrow$ transitions between partial states.

At any given instant in time it is of course true that there is some total state $S_{\Gamma,M}$ that can be composed from the states of all the individual concurrent Pstreams. The definition of splits and merges insures this; at a split every process must go into some Pstream, and concurrent Pstreams resulting from a split are disjoint. Similarly, at a merge, all entering Pstreams become a single Pstream including all the entering processes. Note, however, that since different concurrent Pstreams may take different CFG paths and in general have different number of states, there may only be $\rightarrow$ transitions (in which not all processes change state) between total states in a Pstream execution.

1.8. **CFG vector:** $X(S_{G,J}) = [x_j^p, x_j^q, \ldots, x_j^r]$ for every $p, q, r \in G$, where x_j^p is the CFG node being executed by process p in state $S_{G,J}$. Note that the j (step) values for different processes in the same state may be different from each other since our parallel transition allows processes to advance at different rates.

On repeated execution of the same code, the possibility that each process may take a different path through the CFG and advance at different rates, reaching particular nodes at different step numbers, makes it difficult to see the effects of synchronization code. Graphing the CFG vector, rather than the state directly, has the advantage of displaying the actual code being executed.

2. Synchronization

Synchronization is usually defined through intuition and examples. We here propose a more rigorous definition.

Let t_j^p be the real clock time at which process p reaches state σ_j in its execution. Then synchronization is a pairwise temporal relation between times at which states at different processes execute.

Three binary relations exhaust all the possibilities between a pair of processes:

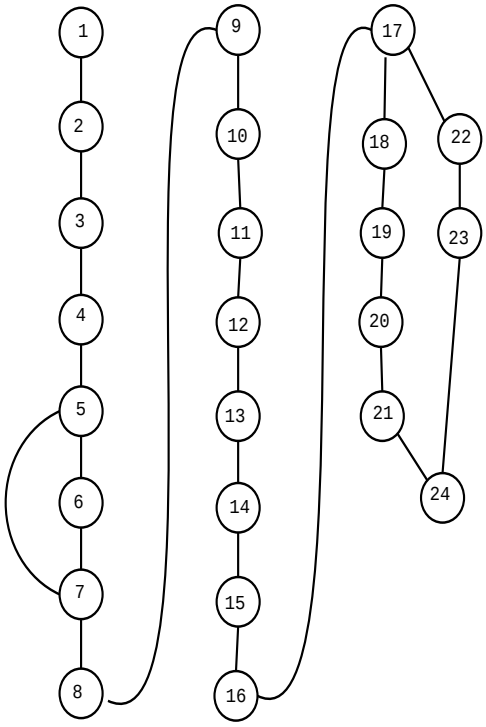
- $EQUAL$: $t_j^p == t_k^q$, exemplified by a *barrier*.

- $NOT-EQUAL$: $t_j^p \neq t_k^q$, exemplified by a mutual exclusion mechanism such as a *semaphore*.
- $GREATER/LESS$: $t_j^p > t_k^q$ or $t_j^p < t_k^q$, exemplified by an ordering relation such as a message with blocking receive, or the *fetch-and-add* instruction of the NYU Ultracomputer.

A synchronization applied to a group of processes establishes some combination of these relations to every combination of process pairs (for example, a barrier synchronization applied to a group G of processes establishes equality between the times of particular states at every process in G).

3. Program CFG

We will use the following control flow graph as an example for visualization:



The test program CFG is non-standard; it is split up into units smaller than basic blocks. This is done so we will be able to display more states in a conceptually very simple program with a simple *loop-until* structure at the beginning and an *if-then-else* structure near the end.

Execution is simulated on a single processor, using a fixed time step. Asynchrony is simulated by adjusting the probability that a process will transition to the next node in the CFG in a single time step. In the synchronous execution, this probability was set to 1. In all the other executions, the probability that a process would transition to the next node in a single time step was set to .9.

4. Visualization

The first question is what to visualize. We would like to be able to see the properties of parallel execution in general, not just the features of a particular execution. Therefore we would like to graph a set, or ensemble, of executions with some common properties (such as the presence of certain kinds of synchronizations). Immediately this leads us to reject the obvious choice of graphing states of an execution versus time; in asynchronous executions, a repeated execution of the same program with the same data may still take a very different amount of time to complete. The same can be said for any particular stage of a repeated execution, which rules out an attempt to normalize the graph to some fixed total execution time.

We choose, therefore, a parametric graph of individual processes in the execution versus other individual processes. The progress of a process from a start to an end state establishes a logical time sequence for that process. Since we take a change of state to occur when a process moves from one basic block in the CFG to another, this logical time sequence is tied to the program code; repeated execution of the same set of states would have the same number of transitions and the same graph, even if the actual execution time is different.

We are still faced with the problem of task parallelism if we want to be able to study the possible executions of the same program text on different data. This leads to the possibility that processes will execute a different number of states to reach the end state (or any intermediate point in the code) on repeated execution of the program.

We choose, therefore, to graph the CFG vector of the program state, rather than the program state directly. This choice ties the visualization to the CFG; as a result the end state is always at a fixed position on the graph. Since the CFG vector is a function of the state, visualizing it gives us a picture of what nodes in the CFG might be executed by different processes at the same time.

A disadvantage of this graph is that, in a more complicated program, such as one with nested loops, repeated display of the same section of the CFG may obscure the details we are interested in. This is not a problem for our simple test CFG, but visualizing a more complicated program may require loop unrolling, artificial limits on the number of times we allow a loop to execute or perhaps a more complicated graph with color variations to denote execution of each loop.

Another problem is the dimensionality of the graph. A parametric graph of the CFG vector requires one dimension per process; an N process execution implies an N-dimensional graph. We assume that, in an ensemble of SPMD executions, any one process could behave like any other, and so an ensemble of two-dimensional slices of the execution is representative of the whole N-dimensional execution. This assumption would not hold for programs that impose some kind of hierarchical structure on the execution based on process i.d., but should be valid for executions in which process i.d. is used mainly to distribute data and any ordering between processes results only from the data.

The following are graphs of two dimensions of a CFG vector - that is, they parametrically display two of N processes in an SPMD execution, considered as a two-dimensional slice of an N-dimensional execution.

In the graphs, process $p = 1$ is displayed horizontally, and process $p = 2$ is displayed vertically, from the top left corner. Coordinates represent node being executed by each process.

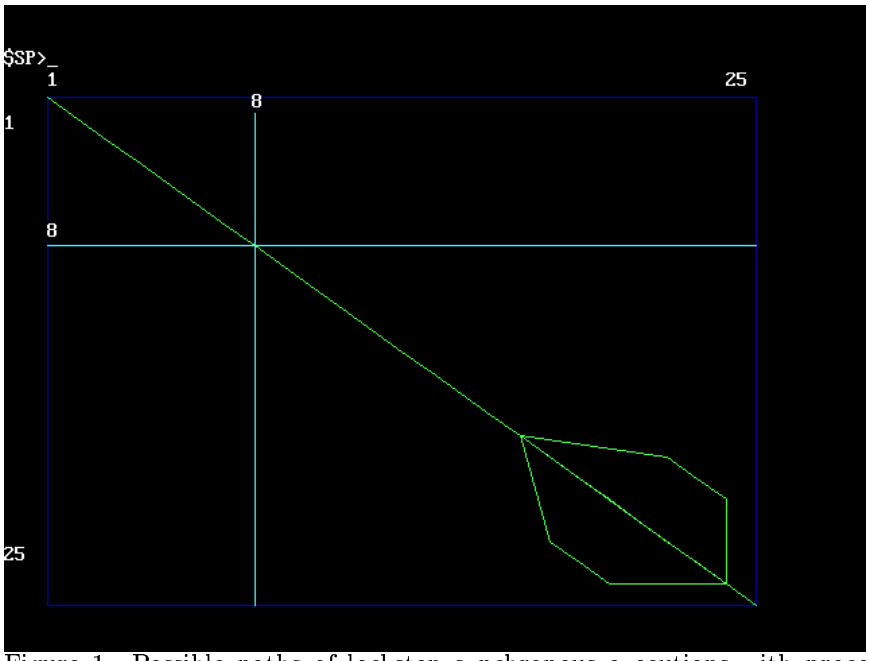


Figure 1: Possible paths of lockstep synchronous executions with procedural parallelism. The straight line at lower right appears only in executions such that both processes take the same path through nodes 17 to 24.

Figure 2: A single asynchronous execution. Loop is visible at top left.

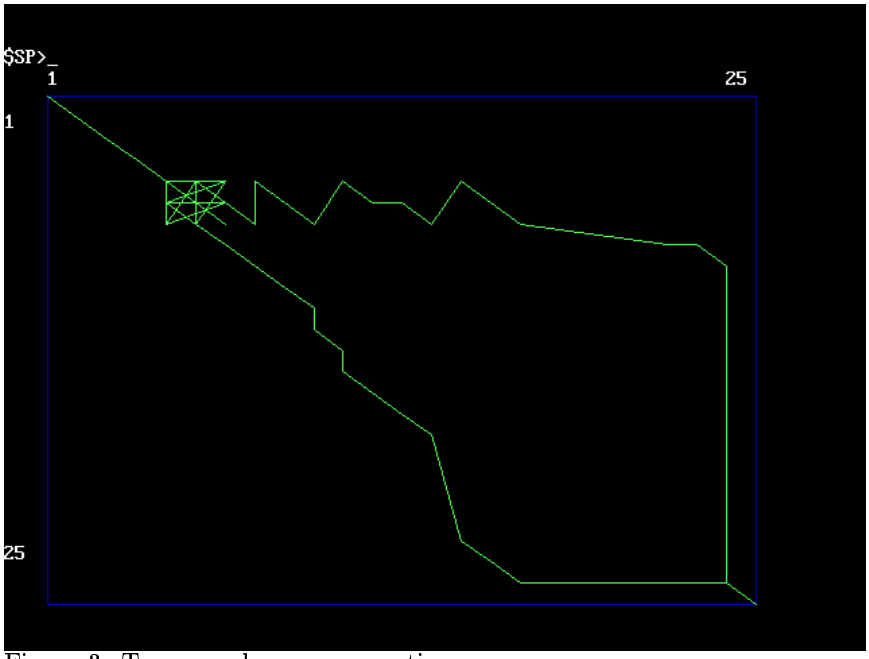


Figure 3: Two asynchronous executions.

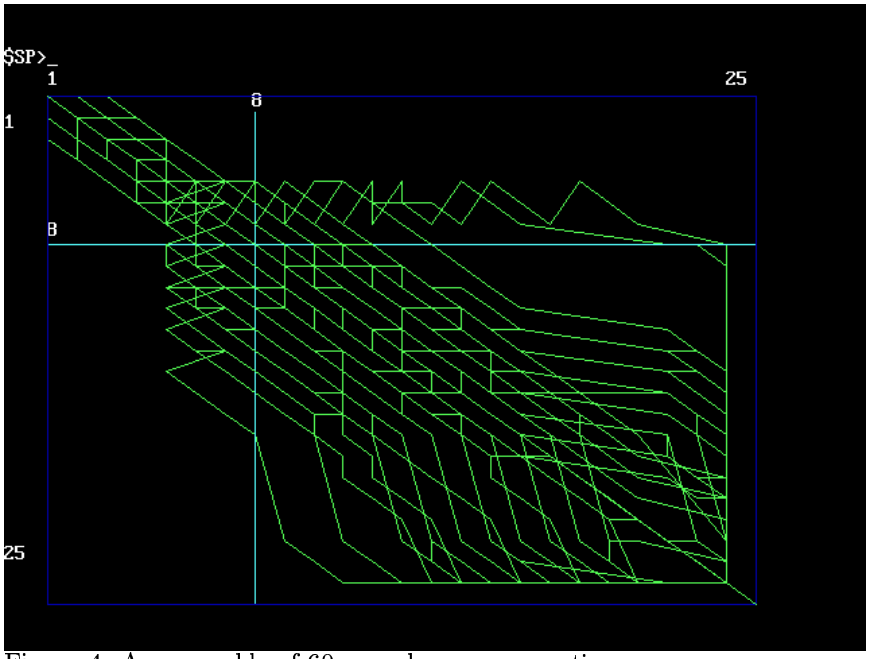


Figure 4: An ensemble of 60 asynchronous executions.

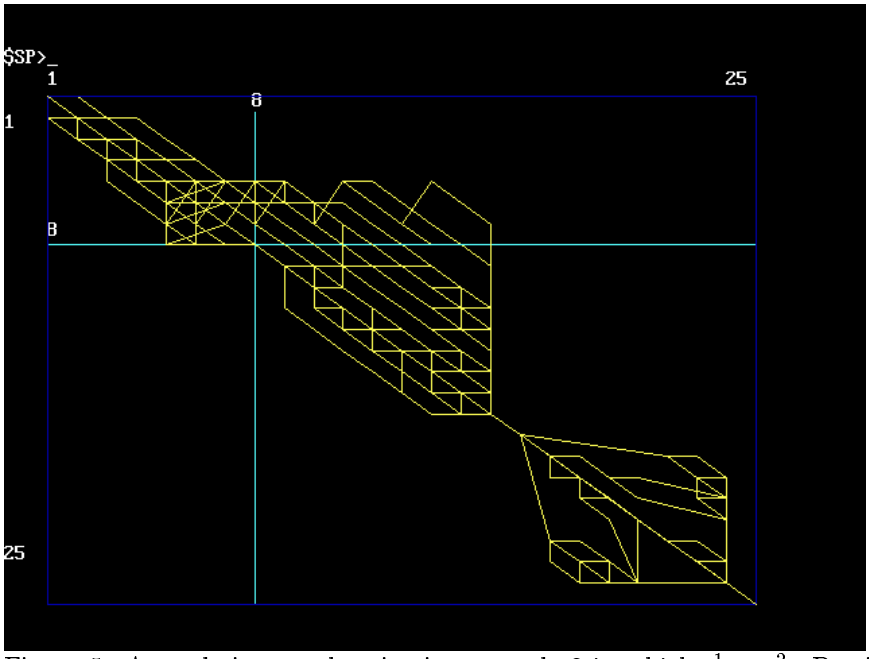


Figure 5: An ordering synchronization at node 8 in which $t^1 \leq t^2$. Barrier at node 16. Repeatable execution characteristics are seen by graphing an ensemble of executions.

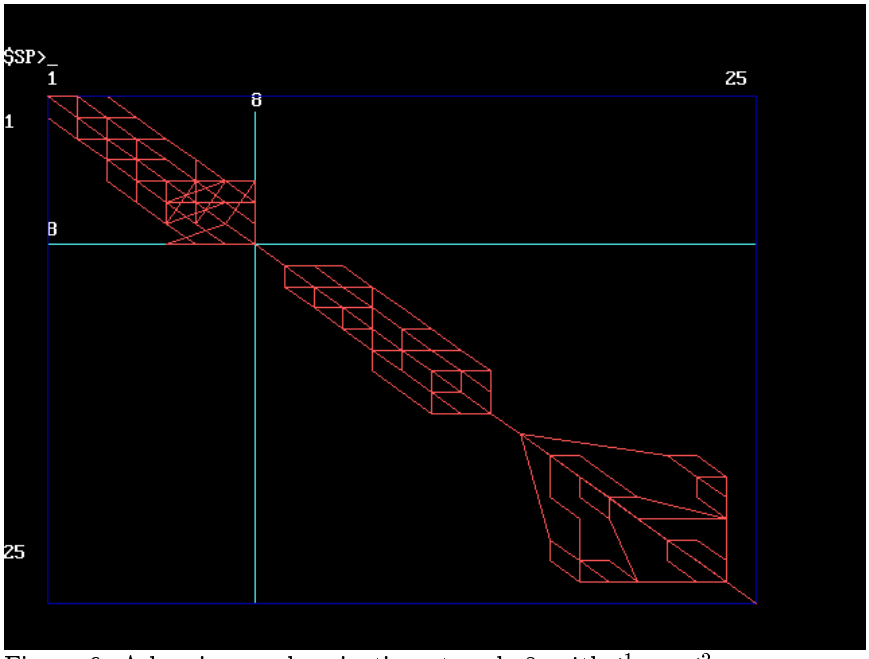


Figure 6: A barrier synchronization at node 8, with $t^1 == t^2$.

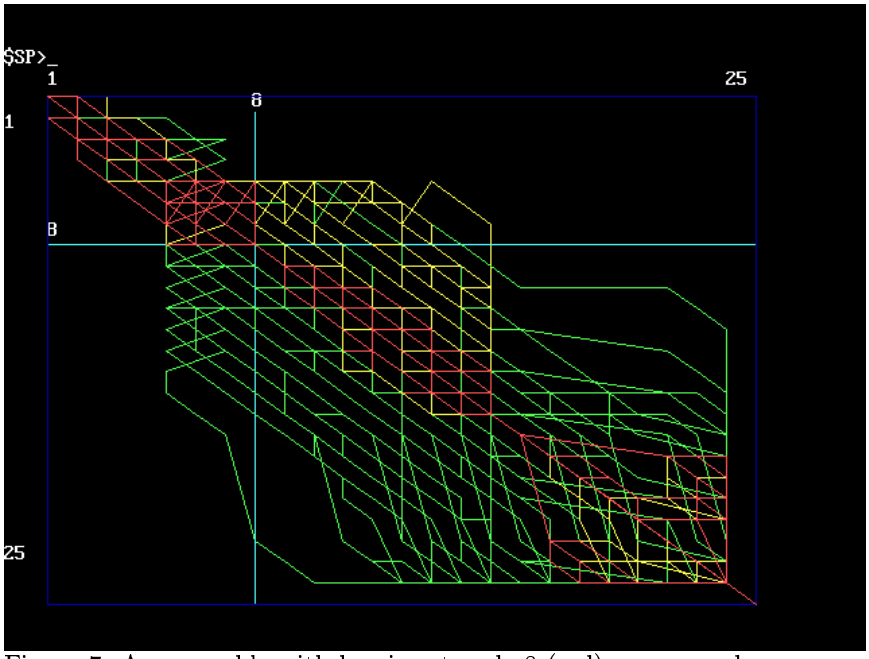


Figure 7: An ensemble with barrier at node 8 (red) superposed on an ensemble with ordering synchronization at node 8 (yellow), superposed on an ensemble with no synchronization.

5. Implications

The following are some results suggested by or supported by the visualization.

5.1. **Models of parallel computation: Pstreams.** Figure 1 displays a set of synchronous executions with lockstep parallelism, in which there is a section of code (an if-then-else) in which processes may take one or another path through the CFG depending on data.

Common theoretical models of parallel execution assume lockstep synchronous processes, requiring that all transitions be of the form $S_{\Gamma,J} \Rightarrow S_{\Gamma,K}$ between total states. Such an execution would look like a straight diagonal line in our visualization. From figure 1 we observe that such models are inadequate to deal with procedural parallelism. We see that an execution with $\Rightarrow$ transitions between total states does not in general exist when processes may take different paths through the CFG.

Pstreams are a theoretical model which establishes an execution based on deterministic $\Rightarrow$ transitions between partial states. Pstreams support process groups implicitly formed by program logic. The model defines split nodes at which subgroups of processes may take different paths through the CFG, and merge nodes at which groups of processes from a previous split come together. Deterministic $\Rightarrow$ transitions can apply to subgroups of processes executing in the same Pstream over a common CFG path.

In SPMD execution, the initial group of processes is a Pstream with group Γ. Splits and merges are the only operations that change Pstream membership, and both satisfy the condition that if the groups entering a split or merge node are

known, the departing groups may be determined. Therefore Pstream process groups are known throughout the execution.

Pstream support is implemented in the SOS (PStreams, Overlapping and Short-cutting) library.

5.2. **Barrier communications and overlapping.** Barriers are based on the $==$ relation, which is transitive and reflective. Therefore no combination of such relations on a group G can deadlock as long as all processes in G are present. Barrier code forces the appearance of a particular state in an ensemble of executions (see figure 6).

Communication with the semantics of a barrier has all the information available at the state determined by the barrier; therefore any communication pattern that is possible at that point in computation can be represented as a barrier communication. Barrier communications, being barriers, are deadlock free if the participating group can be verified. Such verification is provided by Pstream support.

Looking at figure 7, it becomes apparent that the unsynchronized execution is more efficient than either of the executions with synchronizations. Vertical or horizontal paths in the visualization result from some process waiting at a particular node; such paths are a common feature of synchronizations. Synchronization, however, is required to ensure that any particular communication between processes occur.

Overlapping is a technique that uses semantic information from the program code to determine intervals between the updating of a communicated variable and its use. For each communication, a finite state machine is generated as a gatekeeper, and communication is scheduled in the background. If processes are loosely synchronized so that intervals between update and use of a variable at different process overlap in time, then communication scheduled during the overlap does not require that either sender or receiver wait.

Correctness requires that either sender or receiver be made to wait if communication cannot be scheduled during the overlap interval. Barrier communications define a specific parallel state and therefore provide the required semantic information to insure correctness.

Overlapping is implemented in the SOS library.

5.3. **Short-cutting.** A common feature of asynchronous execution, visible in all the figures (except figure 1), is that some processes in a particular execution can be well advanced over other processes. This is particularly evident in the executions with less restrictive synchronization. If a process that has progressed farther can communicate information to a less advanced process, in effect it is communicating information about a possible future.

The less advanced process may be able to use this information to avoid work. This can result, in some special cases, in a parallel execution that does less total work than a corresponding serial execution of the same algorithm; it can potentially result in less work in the average case for the parallel execution. The technique appears to be applicable to backtracking algorithms, and cases where the same correct solution can be reached by different execution paths.

Short-cutting is an implementation of this concept, supported by the SOS library.

5.4. **Continuing work.**

5.4.1. *Visualization:* In this model, processes are graphed parametrically, one dimension per process. Techniques for visualizing ensembles of more than two processes at a time need to be further developed, possibly using three dimensional displays.

The current graphs are only suited to displaying order and barrier synchronizations. Mutual exclusion would appear as a missing state in the graph, and is not easily visible.

Visualization of synchronizations involving only a subset of processes in a group, or involving combinations of different synchronizing relations, need to be tried.

5.4.2. *Work required for synchronization.* Synchronization clearly involves some amount of work; comparing for example figures 1 and 4 make it appear that some large effort must be made to reduce the ensemble of executions to a single possibility. However, the large variation in hardware used for parallel execution, the presence or absence of hardware features like shared memory, a common clock or a synchronization network make it difficult to quantify this work.

Although synchronization hardware may be simulated by messaging, it is not evident that this is any more legitimate or basic than the assumption of some particular hardware.

It may be possible to quantify the work done in principle by synchronization by considering the number of states in ensembles of asynchronous execution, and calculating how many states are eliminated by different synchronizations.

5.4.3. *Synchronization, semantics and overlapping.* Study of the meaning of different basic synchronization types, and combinations between different types, may lead to the application of overlapping techniques to synchronizations that are less restrictive than barriers.

Acknowledgements: Section 2 on Synchronization was developed in collaboration with Pauline Braginton, as part of her Master's research at California State University San Bernardino.

The development of the SOS library, the theory of Pstreams, overlapping and short-cutting was supported in part by the ASCI Flash Center at the University of Chicago under DOE contract B341405.

The support of the National Science Foundation under award 9810708 is gratefully acknowledged.

References

[1] Ernesto Gomez & L. Ridgway Scott, "Overlapping and Shortcutting Techniques in Loosely Synchronous and Irregular Problems" in "Solving Irregularly Structured Problems in Parallel", 5th International Symposium IRREGULAR'98, Proceedings; Springer-Verlag Lecture Notes in Computer Science Vol. 1457

[2] SOS: http://www.csci.csusb.edu/egomez/: links to SOS API, continuing research.