1. Introduction and Motivation

We seek to develop a theory of parallel execution that allows us to describe and prove properties of real parallel execution on real computer systems, without, however, limiting its applicability to specific hardware. We therefore like minimal but realistic restrictions on our model machine.

1.1. Hardware model.

Definition 1.1. Hardware model: multiple, single processor RAM machines connected over a network; each machine has its own memory and clock and interaction between machines is limited to signals or messages sent over the network.

Justification: Since we are talking about parallel computation, our model will have multiple independent computing machines. These could potentially be Turing machines, but for most purposes it will be more useful to assume that they are RAM machines (with Von Neumann architecture). These are computationally equivalent to Turing machines if we allow unlimited RAM.

We need to consider how these machines are linked and synchronized with each other. It is possible to have multiple machines on a single chip or board, possibly accessing the same memory. However, it appears that there are physical limits to cooling and circuit construction that impose a fairly hard upper bound on the number of processors we can link in this way (although this upper bound will likely shift as technology improves). It is unclear to what extent we can assume that all machines run off a single clock, even if all processors are on the same board but not on the same chip. Here a hard physical limitation is the speed of light, which limits how closely we can synchronize physically separated events, and limits causal connections and information transfer to lower speeds.

We can, however, always link multiple machines, each with its own local memory and clock, on a network. The clocks can be synchronized through sequences of signals on the network (although the possibility of drift means the synchronization needs to be periodically verified or reset). Such a network can scale to an arbitrary number of processors, although we need to accept varying message delays between different nodes. For the foreseeable future it is likely that we will actually have a hierarchical arrangement, in which multiprocessor nodes with shared memory and a single clock are members of a larger network of many such nodes (note that this still applies even if the multiprocessors are inside a GPU - however GPU computation may force us eventually to build two levels into the model, with multiple stream
processors with shared memory and clock in a GPU communicating to other such sets of processors/processes).

Given a network of multiprocessor RAM machines, we consider a parallel execution to consist of multiple processes running on multiple processors, such that the individual processors are either on the same RAM machine, or on different machines. Since the network will scale to larger numbers than any individual machine, we expect that the typical interaction between processes will be over the network between different RAM machines. Even though processes running on the same RAM will have access to shared memory, most of the memory available to a parallel computation will be accessible only through the network.

Processes running on different machines will be subject to different clocks, possibly different execution speeds and effectively random variation in message delay. Even processes running on the same machine, with the same clock, will be subject to interruptions from other tasks and are therefore likely to run at different speeds.

We find that process execution on different nodes, with separate clocks, distributed memory and speed variation between processes to be more characteristic of a real parallel execution than the shared memory and single clock available on a multiprocessor machine. We further believe that the constraints imposed by physics and engineering are such that separate clocks, distributed memory and network communications are likely to be characteristics of real parallel computation systems with large numbers of processors in the future.

1.2. Process model.

Definition 1.2. Process model: a parallel execution consists of some number $N$ of processes, running on a network of $M$ RAM machines, with $N \leq M$ and not more than one process per machine. We assume a random variation in execution speeds between different processes, and within the same process at different times.

We will consider a situation in which each RAM is running a single process that is part of a parallel computation. Realistically each machine will also have a runtime system and an operating system, which will occasionally interrupt the compute process. We will ignore these interruptions and consider any delays caused by operating or runtime system as equivalent to clock or speed variations between different machines.

The result of interruptions to process execution, clock or speed variations is that, over any long computational interval, processes running on two different machines will execute instructions at different average rates. Therefore the rate of progress of even two perfectly load-balanced processes will be different. Specifically, if a process $p$ has taken $N$ steps in its computation in time interval $t$, an identical process $p'$ running on a different machine cannot be guaranteed to take the same $N$ steps during the same interval $t$. Allowing for runtime and operating system delays we find that even repeated execution of $p$ on the same machine for the same interval $t$ will not guarantee execution of the same $N$ steps. (This is verified by practical experience; for instance running the same benchmark multiple times on the same machine will not yield identical results for each run).

1.3. SPMD execution model.

Definition 1.3. SPMD stands for Single Program, Multiple Data. Specifically, we have multiple copies of the same program text, running concurrently, each
copy having a unique identifier and potentially acting on different data. We will use two variants of this model: Static - in which the number of processes is set at the start of execution and fixed thereafter; and Dynamic - in which processes may be added or deleted during execution. A defining characteristic of SPMD is that control logic acting on data-dependent predicates can cause processes to execute different code sections concurrently (Procedural parallelism).

Justification: the most general parallel execution model is MIMD, in which different program texts, each with its own data, execute concurrently and interact. It is evident that our hardware and process models support MIMD execution (because we have independent processes running on different computers). However, full MIMD makes it difficult, if not impossible, to say anything about the relation between execution and code, and in particular it hinders investigation of the relation between code and desirable properties of execution. We therefore impose the restriction of concurrent execution of the same program text; since this allows procedural parallelism, we have in fact the same freedom to execute different code at each process that is allowed by MIMD. However, in SPMD we begin execution as a set of processes with the same code, and this allows us to establish an initial link between all processes and then investigate how the relation between processes changes during execution as a consequence of specific code.

We choose to investigate SPMD executions because we believe it is the most general execution model that still allows us to easily link parallel execution to program code.

2. Definitions

A basic block is a set of statements that are always executed together (straight-line code). As such it is a unit of program code that can be regarded as a function that, given specific input (contents of memory, data read) produces specific output (contents of memory). No larger program unit can be guaranteed to have this property with regard to memory contents, since no larger program unit can be guaranteed to always execute the same instructions. The Control Flow Graph is an abstraction that allows us to represent a program in terms of its basic blocks; we will use this abstraction to build our notions of state and program execution (see notes on control flow graph and basic blocks).

Given a program with a CFG \( <V, A, s, E> \) where \( V \) is the set of basic blocks which are vertices, \( A \) is the set of arcs, \( s \in V \) is the start block of the program \( E \subseteq V \) is the set of end blocks, we define:

**Definition 2.1.** Serial state: \( \sigma_i = (x_i, D_i, M_i) \) where \( i \) denotes the step in program execution, \( x_i \) is the basic block executed at step \( i \), \( M_i \) is the contents of memory after execution of block \( x_i \), and \( D_i \) is data input when block \( x \) is executed. Take \( M_{0i} \) to be the contents of memory at the start of block \( x_i \); then \( M_i = x_i(M_{0i}, D_i) \). The memory contents at the end of block \( x_i \) are a function of the memory contents at entry to block \( x_i \) and any data input in block \( x_i \), and this function is defined by code in \( x_i \). Where no confusion is possible, we omit the \( D_i \) term for reasons of brevity (since input \( D_i \) is implicit in the execution of \( x_i \)).

We now define a serial transition:

**Definition 2.2.** Serial transition: \( \sigma_i \rightarrow \sigma_{i+1} = (x_i, D_i, M_i) \rightarrow (x_{i+1}, D_{i+1}, M_{i+1}) \) such that \( x_{i+1} \) is the block to which execution transitions after execution of block
Given the transition, we now define execution as a series of transitions:

Definition 2.3. Serial execution: \( e = σ_0 → σ_1 \rightarrow \ldots \rightarrow σ_N \) for a non-divergent execution (that is, one which terminates) in \( N \) steps, where \( σ_0 = (s,M_0) \) and \( s \) is the start block. We will often designate the final state as \( σ_e = (x_e,M_e) \) where \( x_e \in E \), one of the end blocks, and the subscript \( e \) here denotes the end state of the execution. We may abbreviate this \( e = σ_s →^* σ_e \) to denote 0 or more transitions between a start state and an end state. For a possibly divergent execution we will write \( e = σ_s →^* σ_e |⊥ \), where \( ⊥ \) denotes divergence (failure to halt). We denote an execution on specific data \( D \) by \( e_D \).

It is straightforward to show that a serial execution is deterministic, depending only on the code.

Theorem 2.4. Serial execution is deterministic if random elements are not deliberately introduced in code.

Repeated execution of the same code with the same sequence of data inputs results in the same serial execution; that is one which passes through the same set of states to compute the same final state of memory.

Proof. Let \( σ_i \) be a state 2.1. In serial execution there is a transition \( σ_i → σ_{i+1} = (x_i,M_i) → (x_{i+1},M_{i+1}) \). The contents of memory at the end of execution of block \( x_i \) are given by \( M_i = x_i(M_{i-1},D_i) \), where \( M_{i-1} \) is memory contents at the beginning of block \( x_i \). \( D_i \) is any data input during execution of \( x_i \). If \( x_i \) contains no explicitly non-deterministic code, and since \( D_i \) is fixed by assumption, then \( M_i \) depends only on \( M_{i-1} \). The choice of block \( x_{i+1} \) to be executed next is determined by the memory contents at the end of block \( x_i \); since these contents are uniquely determined, so is the block \( x_{i+1} \). Therefore the transition \( σ_i → σ_{i+1} \) has only one possible outcome and is deterministic.

Given a sequence of data inputs \( D \), an initial state \( σ_0 = (x_0,M_0) \) the transition out of \( σ_0 \) is deterministic. It follows by induction that all subsequent transitions are deterministic. Therefore on repeated execution with the same input \( D \), all states and computed results are the same.

Take now a set of processes \( Γ \) and a data set \( Δ = \bigcup^{p=0,M} D_p \). It is tempting to define a parallel execution as a set of serial executions \( E_{Γ,Δ} = \{ e_D^p \} \). The problem with this approach is that it does not lead to any useful way of defining a parallel state.

2.1. Parallel state and transition. We instead start by defining a parallel state as a collection of individual process states:

Definition 2.5. Parallel SPMD state:

\[ S_G,J = \{ σ_p^| p ∈ G, j ∈ J, G ⊆ Γ \} \]

\( G \) is an ordered set of processes and \( J \) is a multi-index that lists the step number of the state of each process in \( G \) in the same order as \( G \). \( Γ \) is the ordered set of all processes. For an execution \( E_Γ \), if \( G = Γ \) then we call \( S_{Γ,J} \) a total state. Note that \( S_G,J \) is a collection of serial states such that each state is selected from the execution of a different serial process.
We note that $S_{G,J}$ can describe a state that does not include all the processes in an execution. It is evident that for $S_{G,J}$ to exist, there must be $S_{\Gamma,K}$ such that $G \subseteq \Gamma$ and $J \subseteq K$. We nevertheless need partial states $S_{G,J}$ because program code may imply or force relations including only a subset $G$ of all processes.

2.1.1. The need for partial states. A defining feature of a partial state is a set $G$ of processes that is a subset of the set of all processes $\Gamma$. However, if such a process set $G$ cannot affect program execution, it is meaningless. Therefore we will normally define process sets, explicitly or implicitly, in program code. We consider several examples of this:

A point to point synchronous communication is a relation between two processes and can thus be described as a partial state including only those two processes. The same data transfer could be accomplished by a barrier followed immediately by a `get` at the receiver process or a `put` at a sender process (the barrier is necessary to ensure determinism). However, the barrier includes all processes and therefore enforces a total state.

A different example is illustrated by code of the form `if (x) then {A} else {B}`. Let $x$ be a non-uniform predicate; that is, one that may have different values at different processes (in Scott, Clark + Bagheri a uniform value is said to be global, a non-uniform value is called local - pg 195); let $A$ and $B$ be code blocks. This code creates two subsets of processes, $G_A$ and $G_B$, executing blocks $A$ and $B$ respectively. Processes in the same subset are implicitly related by the fact that they are executing the same code. This has significance for the program; for example if we wish to have a barrier that includes all processes in $G_A \cup G_B$, we need to insert code into both block $A$ and block $B$; whereas if we only care about synchronizing processes in $G_A$ then inserting code into block $A$ is sufficient. It is useful here to describe partial states with process sets $G_A$ or $G_B$ because code that appears only in block $A$ or block $B$ can only directly affect $G_A$ or $G_B$, respectively.

In the above examples, process sets of partial states are implicitly defined by the execution of specific code such as a communication statement or a control statement. We may also admit a process set defined by a data structure, such as an MPI communicator.

We note further that the code or data structure that defines a process subset and therefore a partial state may act as an upper bound on the processes actually in the partial state. Consider a set of partial states defined by an MPI communicator. The code that establishes such a communicator is local to each process; following our process model (def. 1.2) it is clear that this code will be executed at different times in different processes; in the time interval between the first process defining the communicator and the final process defining the communicator, the actual process subset is smaller than explicit subset definition in the MPI communicator.

Note 2.6. When comparing two parallel states $S_{G,J}$ and $S_{\Gamma,K}$, $J,K$ are in fact multi-indices, not sets ($G, \Gamma$ are ordered sets). We extend set notation to denote inclusion of one multi-index in another, in the appropriate order. That is, if for example $J$ denotes the step numbers of processes in $G$, then $J \subseteq K$ denotes the presence in $K$ of the same indices in $J$, in the positions in which processes in $G$ appear in $\Gamma$. We will use set notation in this sense when referring to multi-indices.

2.1.2. Functions of parallel state. We will need several functions of a parallel state:
\[ \beta(S_{G,J}) = X_{G,J} = \{ x^j_p \mid p \in G \}, \] that is, the ordered set of basic blocks which the processes in \( G \) are executing at step \( j \in J \); and

\[ \mu(S_{G,J}) = M_{G,J} = \{ M^j_p \mid p \in G \}, \] that is, the ordered set of memory contents of processes in \( G \) at step \( j \in J \).

These functions for a parallel state are equivalent to functions of a serial state that select the basic block or the contents of memory.

2.1.3. Extension to definition of basic block. Take state \( S_{G,I} \) which is consistent with a total state \( S_{\Gamma,I} \). Let \( M_{G,0} \) denote the state of memory at each process in \( G \) on entry to the individual process states identified in \( I \), and let \( D_{G,I} \) be the set of data inputs read by each process in \( G \) in their respective states in \( I \) (we define \( M_{\Gamma,0} \) and \( D_{\Gamma,I} \) analogously).

We wish to define the state of memory in a particular process in the same way as we did in the definition of serial state 2.1. There we had:

\[ M_i = x_i(M_{0,i}, D_i) \]

Now, since there may be communication between processes, we have that \( M^P_i \) is a function of not only \( M^0_i \) as before, but also potentially a function of the memory at all processes in \( G \) or even in \( \Gamma \). Further, if there are communication statements in block \( x^p_i \) (or, indeed, if there are put statements in any block \( x_i \in X_{\Gamma,I} \)), then \( M^P_i \) is potentially a function of memory during the execution of any or every such block; that is we have to consider memory states intermediate between \( M_{\Gamma,0} \) and \( M_{\Gamma,I} \). We have:

\[ M^P_i = X_{\Gamma,I}(M_{\Gamma,0}, M \leq t M_{\Gamma,I}, D_{\Gamma,I}) \]

where the \( M \leq t M_{\Gamma,I} \) denotes memory at some time less than or equal to the time when \( M_{\Gamma,I} \) has been computed. This has the added difficulty that every \( M^P_i \in M_{\Gamma,I} \) must be computed the same way.

We may reduce the complexity of this expression by modifying the standard definition of basic block (see Aho, Sethi and Ullman; Compilers - Principles, Techniques and Tools for the standard definition). To the list of "leaders" - statements that define the start of a basic block - we add communication statements. That is:

**Proposition 2.7.** Leaders that define the start of a basic block include every statement that can transfer information from one process to another process. Therefore we can have only one communication statement per basic block.

The execution of one basic block defines a serial state; transfer of execution from one block to another defines the serial transition (def. 2.2). If every communication statement begins a basic block, then any information transferred refers to the contents of memory at the start of the block \( M^0_i \), and we can write:

\[ M^P_i = X_{\Gamma,I}(M_{\Gamma,0}, D^P_i) \]

which looks like our expression for the memory in a serial state, except that now the memory in each state may depend on the memory at other processes and the code at other processes. Note that \( M^P_i \) now only depends on \( D^P_i \), since any data read at other processes may only be transferred at the start of each block and is included in \( M_{\Gamma,0} \).

If we further restrict ourselves to two-sided communications (e.g. message passing), then it is only possible to receive information from other processes at the start of a block. Therefore the memory at the end of execution of a serial basic block depends only on the code in that block, although it may still depend on memory at other processes.
\[
M_p^p = x_p^p(M_{T,01},D_p^p)
\]

2.2. Parallel transition. We now define a general parallel transition from one state to another:

Definition 2.8. A parallel transition \( S_{G,J} \rightarrow S_{H,K} \) exists between two states such that:

i. \( G \cap H \neq \emptyset \) (there are processes in common between both states).

ii. There is at least one serial state \( \sigma_j \in S_{G,J} \) and at least one serial state \( \sigma_k \in S_{H,K} \) such that there is a serial transition \( \sigma_j \rightarrow \sigma_k \). (at least one process takes a step).

iii. There are total states \( S_{G,J} \subseteq S_{T,L} \) and \( S_{H,K} \subseteq S_{T,M} \) such that \( \forall p \in \Gamma \), if \( s_j^p \in S_{T,L} \) and \( s_k^p \in S_{T,M} \), then \( 0 \leq k - j \leq 1 \). That is, states of individual processes that appear in both total states are separated by not more than one step. (Therefore there is a transition \( S_{T,L} \rightarrow S_{T,M} \) between total states that is compatible with or includes \( S_{G,J} \rightarrow S_{H,K} \)).

A parallel transition \( S_{T,J} \rightarrow S_{T,K} \) between two total states is called a total transition; if either or both states are partial we may refer to it as a partial transition.

We denote a parallel transition in which a specific process \( p \) undergoes a serial transition by a subscript on \( \rightarrow \), as follows: \( S_{G,J} \rightarrow_p S_{H,K} \).

We note that partial states are defined by code, however code is necessarily local to each process. Membership in a state is therefore a local view. When \( S_{G,J} \rightarrow_p S_{H,K} \) occurs, \( p \) views itself as being in \( S_{H,K} \) but it is possible that some other process \( q \neq p \in G \cap H \) has not transitioned and still views itself in \( S_{G,J} \). However, neither \( S_{G,J} \) nor \( S_{H,K} \) are complete, in the sense that neither group \( G \) nor group \( H \) in fact include all the specified processes. So what \( p \) is doing is asserting a state \( S_{H,K} \) which may or may not come into existence. (If communication takes place in a state, then it may require that the state actually exists, so it is necessary to verify the assertion.)

We may view \( H \) as defined by some code that exists in \( \beta(S_{H,K}) \), whereas \( \beta(S_{G,J}) \) contains code that defines \( G \). Then \( S_{G,J} \rightarrow S_{H,K} \) actually includes multiple transitions, between two end states with process sets \( G \) and \( H \), such that at any given time the actual process sets are \( G' \subseteq G \) and \( H' \subseteq H \), with states \( S_{G,J} \) and \( S_{H,K} \) existing concurrently. We of course need to consider the possibility that \( S_{H,K} \) never in fact exists, but that processes that enter \( S_{H,K} \) transition out of that state before all processes in \( S_{G,J} \) have transitioned into \( S_{H,K} \) (and where, in this discussion, \( S_{H,K} \) encompasses multiple states that meet the condition \( H' \subseteq H \)).

The general parallel transition is not deterministic. Even if each individual serial transition is deterministic, absent restrictions on the timing of each execution we do not know which particular process or processes will transition at any given time.

In general, given \( M = |\Gamma| \) (cardinality of \( \Gamma \)) processes the step numbers of which are given by a multi index \( J \), there are \( 2^M - 1 \) possible serial transitions (the power set, or set of all subsets of \( J \) less the empty set). If we are dealing with partial states, we need to add the that each of the \( M \) processes is also asserting a change in the process set it belongs to. If we allow a single process to belong to multiple process sets, then the possible changes in process subsets are given by the power set of \( \Gamma \) (the set of all subsets), the cardinality of which is also \( 2^M - 1 \). We may restrict our model to allow membership only in a single process subset; in this case there
are at most $M$ different subsets of $\Gamma$ (the case in which each process is a subset with one member).

2.3. **Parallel execution.** We now define an SPMD parallel execution. We will restrict ourselves initially to the static SPMD model, in which there is a fixed number of processes; that is, $\Gamma$ is constant.

**Definition 2.9. Parallel SPMD total execution:** Given a set $\Gamma$ of serial processes numbered 0 to $M$ ($M$ is a constant for static execution), and an external data set $\Delta = \bigcup_{p=0}^{M} D^p$, we have a parallel execution defined by total transitions:

$E_{\Gamma,\Delta} = \Delta_{0} \rightarrow S_{\Gamma,0} \rightarrow S_{\Gamma,1} \rightarrow \ldots \rightarrow S_{\Gamma,End}$

$0$ denotes the multi-index in which every process in $\Gamma$ is in its initial state, and $End$ denotes the multi-index in which every process is in its final state.

We may also describe an execution in partial states:

**Definition 2.10. A Parallel SPMD partial execution** that does not diverge is given by a directed acyclic graph $E_{\Gamma,\Delta} = <V,A,S_{\Gamma,0},S_{\Gamma,End}>$ in which the nodes $V$ are partial parallel states, the arcs $A$ are parallel transitions $S_{G,J} \rightarrow S_{H,K}$ and $S_{\Gamma,0}, S_{\Gamma,End} \in V$ are, respectively, start and end nodes. Data is again given by $\Delta = \bigcup_{p=0}^{M} D^p$.

The start node is a total state, since subsets have not been established until code is executed. The end node is also a total state, since in a successful execution in the static model, all processes in $\Gamma$ must reach the end state.

We note, however, that for either a partial or a total execution to exist, additional conditions must hold.

**Condition 2.11. Absence of deadlock:** Given a partial state $S_{G,J} \notin S_{\Gamma,End}$ (not an end state or part of one), and given a process $p \in G$ which is not in an end state of its serial execution, then there is a transition $S_{G,J} \rightarrow_p S_{H,K}$ for $p$ out of $S_{G,J}$.

We also have:

**Condition 2.12. No divergence:** there is a finite series of transitions $S_{\Gamma,0} \rightarrow^*_p S_{\Gamma,End}$ for every $p \in \Gamma$.

We have noted that the parallel transition is not deterministic. As a result, $E_{\Gamma,\Delta}$ is also non-deterministic. In fact, if we only limit ourselves to deterministic serial processes but impose no timing restrictions, and if each of $M$ processes in $\Gamma$ takes $N$ steps to reach $End$, then we have $(2^M - 1)^N$ possible executions with the same $\Delta$ and $\Gamma$.

**Definition 2.13. Ensemble:** We call any collection of executions of a particular program an ensemble of executions $\Theta$. We will say that a **data-process ensemble** $\Theta_{\Gamma,\Delta} = \{E_{\Gamma,\Delta}\}$ is a set of executions of the same program, with the same data set $\Delta$ and the same set of processes $\Gamma$. We distinguish different specific executions in the ensemble by a superscript $\varepsilon$.

We may impose further restrictions on ensembles to describe particular types of execution. The study of ensembles of executions rather than particular executions, allows us to study those features of parallel execution that characterize a program, and distinguish them from accidental properties that may not be repeated.
2.4. Properties of parallel execution: We are specifically interested in two properties of a data-process ensemble (same program, same data in the same sequence) $\Theta_{\Gamma,\Delta}$:

2.4.1. Absence of deadlock. Absence of deadlock is necessary for the execution to exist (that is, reach completion).

There is no universal agreement on what is meant by deadlock, although there is general agreement on the concept that it is a condition in which a process waits forever for some external resource (such as a message).

The standard definition of deadlock follows from the results derived by Coffman, Elphick and Shoshani in *System Deadlocks*, Computing Surveys, vol. 3, June 1971, citing four conditions that are required to hold simultaneously for deadlock to exist in a system: 1) mutual exclusion, 2) hold and wait, 3) no preemption and 4) circular wait. Many authors cite a circular dependence (condition 4) as the key condition.

We prefer a definition that does not specify why the resource does not become available (which also covers conditions sometimes known as live-lock):

**Definition 2.14. Deadlock:** a condition of a parallel state $S_{G,J}$ in which there is at least one process $p$ with state $\sigma^p_j$ which is waiting for some external event that never occurs and therefore never carries out a transition $\sigma^p_j \rightarrow \sigma^p_j+1$.

In distributed execution, the external event is usually a message sent by another process. Note however that this message could be a protocol message, so sender processes can also be affected by deadlock if the receiver never allows the send to proceed.

Given this definition, absence of deadlock then implies that each serial process in a distributed execution gets whatever messages or other signals it needs to continue, and will only fail because of some problem internal to the process. This definition essentially isolates the new features introduced by parallel execution from the problems already present in standard serial execution.

2.4.2. The loop. It is possible to have a sequence of partial transitions each of which appears locally correct, but which leads to a state that is incompatible with a total transition. This may occur because code that imposes partial states is generally unable to verify compatibility with total states.

Consider a parallel transition $S_{G,J} \rightarrow S_{H,K}$ following definition 2.8.

We here consider the consequences of omitting condition iii.

Let $\Gamma = \{0,1,2\}$; the set of all processes. Let $(i,j,k)$ be a multi-index describing a particular state such that the indices denote the step (state) number for processes 0, 1, 2 respectively. Let * in place of any index denote a process not in the state. For example, $(0,1,*) \rightarrow (1,*,0)$ describes a transition $S_{G,J} \rightarrow S_{H,K}$ where $G = \{0,1\}, J = (0,1); H = \{0,2\}, K = (0,1)$. Since process 0 is in $G \cap H$, and process 0 steps from 0 to 1, conditions i and ii are met.

Now consider the following sequence (here shown as a column for ease in aligning processes and sequential state changes):

**Case 1. The Loop**
1. $(0,1,*)$
2. $(1,*,0)$
3. $(*,0,1)$
From the global perspective, $2 \rightarrow 3$ and $3 \rightarrow 4$ are impossible - $1$ and $4$ are the same state so there is no transition between them, and $3$ is not compatible with any global state that could be created by any set of transitions from $1$. However, each individual transition from one state to the next meets conditions i and ii. We call this situation "the Loop", because it suggests a loop in time, which is obviously impossible to execute but which is implied by the first two conditions of a parallel transition.

We need to consider if "the Loop" can correspond to real code - that is, can we inadvertently write such a case?

In our example, we are specifying states of two processes. A real situation in which such a state arises is in an MPI synchronous send and receive between a pair of processes. For example, executing:

```
if(ME==0) MPI_ssend(x to 1) if(ME==1) MPI_srecv(x from 0)
```

implements $(0,1,*),$ if we take execution of the first line followed by the second line as a state transition so both processes start at state $0$, process $0$ sends in state $0$ but process $1$ transitions to state $1$ to execute the corresponding receive. We do not care what process $2$ is doing while this happens, so we do not constrain its execution in any way - it is not part of the state so we denote the sequential state of $2$ by $*$. State $(1,*,0)$ is similarly implemented by a synchronous send from $0$ to $2$, and state $(*,0,1)$ by a synchronous send from $1$ to $2$.

If we write the code in the order i->ii->iii we get:

```
Case 2. Code for the Loop - first attempt
1.a if(ME==0) MPI_ssend(x to 1)     
1.b if(ME==1) MPI_srecv(x from 0)   
2.a if(ME==0) MPI_ssend(y to 2)     
2.b if(ME==2) MPI_srecv(y from 0)   
3.a if(ME==1) MPI_ssend(z to 2)     
3.b if(ME==2) MPI_srecv(z from 1)   
```

We here assume that each pair of send/receive statements transfers a different variable, to avoid any possible problems with data dependences.

However, the code in 2 does not properly represent the sequence described by 1 because process $1$ is supposed to send to $2$ in its state $0$, before it receives from $0$ in its state $1$. We can remedy this by moving statement $3.a$ to a position before 1.b. Note that we are not prevented from doing this by any evident program or data constraints, since $3.a$ is independent of all prior statements in this sequence.

We now have:

```
Case 3. Code transformed to match 1
1.a if(ME==0) MPI_ssend(x to 1)     
3.a if(ME==1) MPI_ssend(z to 2)     
1.b if(ME==1) MPI_srecv(x from 0)   
2.a if(ME==0) MPI_ssend(y to 2)     
2.b if(ME==2) MPI_srecv(y from 0)   
3.b if(ME==2) MPI_srecv(z from 1)   
```

This code is not obviously impossible to execute. No data dependences are violated, and all sends appear before the corresponding receives. Further, there can
be no cycles of dependences between multiple processes because $p=0$ only sends, and $p=2$ only receives. 4 depicts the pattern of sends and receives.

Case 4. Pattern of sends and receives in 3

$p0 \rightarrow p2$

$p0 \rightarrow p1 \rightarrow p2$

However, because the sends and receives are synchronous, we force particular states to exist, and this must be reflected in the execution of code. In particular, the following statements pairs must occur simultaneously:

Case 5. Simultaneous statement pairs

$s1 \ [1.a@0 \text{ and } 1.b@1]$

$s2 \ [3.a@1 \text{ and } 3.b@2]$

$s3 \ [2.a@0 \text{ and } 2.b@2]$

Now, comparing 1.1 with 3 we can make the following statements about time sequences:

Case 6. Temporal dependences

$s1 < s2$ because statement 1.a is before 2.a at process 0

$s2 < s3$ because 2.b is before 3.b at process 2

$s3 < s1$ because 3.a is before 1.b at process 1

(Time sequence is given by pairs of communication statements that run at the same process).

The sequence in 6 is a cycle which will deadlock, but it is not a cycle between processes or a cycle of data dependences - it is a cycle between parallel states, reflecting an execution of the form:

$s1 \rightarrow s2 \rightarrow s3 \rightarrow s1$; that is, 'the Loop' from 1 in which the first state repeats.

Condition iii from the definition 2.8 of a parallel transition would have avoided the problem - it is clear from 1 that state 3 is not compatible with any set of total transitions from state 1. However, the given code explicitly defines partial states and says nothing about total states. This is in general the case for code - we impose conditions on execution (i.e. $if(function - of - ME)$) which create partial states by selecting some processes to perform actions that are not performed by other processes. However, correctness of a series of partial transitions depends on their correspondence to total transitions, which are visible only in a global view of execution.

2.4.3. Determinism in parallel execution. Determinism is desirable in that we generally want computation to be repeatable given the same input data. In some cases we are interested in non-deterministic computation if we can guarantee that its results are approximately correct, or if we can guarantee one of several different correct results. We will not consider deliberately introduced non-determinism here, except to note that it generally is overlaid on an otherwise deterministic program.

We can have several variations on the definition of determinism for parallel execution. A definition along the lines of 2.4 would require that the particular sequence of parallel states be repeatable.

We will call this strict determinism.

Definition 2.15. Strict Determinism: Given a specific external data set $\Delta$ and a group of processes $\Gamma$, the execution $E_{\Gamma,\Delta}$ always follows the same sequence of parallel states 2.5 and ensemble $\Theta_{\Gamma,\Delta}$ has only one member.
In order for strict determinism to hold, we require coordination between every process at every step. The most obvious way of achieving this is what we term lock-step synchronization, in which every process takes a step at the same time. Note, however, that this is not the only possibility - consider for example an execution in which some particular processes take steps twice as fast as other processes, and this difference in rates is repeatable. It is clear, though, that coordination between steps at different processes is required to ensure this repeatability.

We may loosen the definition somewhat if we require that each process be individually deterministic as in 2.4, but do not require that steps at each process be coordinated with the others. Strict determinism does not hold because the sequence of parallel states may be different, even with the same data:

**Definition 2.16. Process Determinism:** Given a specific external data set $\Delta$ and a group of processes $\Gamma$, each process $p$ in the execution $E_{\Gamma, \Delta}$ always follows the same sequence of serial states 2.1.

It is evident that an execution that is strictly deterministic is also process deterministic, but the converse does not hold. We further note that process determinism may require restrictions on communications, since data transfers between processes modify the memory at each process and therefore the serial state.

### 3. Synchronization

We use the term synchronization to refer to any coordination between processes. This implies a relationship between times at different processes, as measured by clocks.

There are many synchronization mechanisms in use to enforce this coordination; however they all must be enforced either in hardware, software or some combination of the two. (There are theoretical models that assume synchronization. We comment that to implement any of those models requires extra code or extra hardware that is not part of the model).

We define synchronization as follows:

**Definition 3.1.** A synchronization is a relationship in time between specific states of particular processes in a set of processes, enforced by code which establishes the same relationship over repeated executions of a program.

Note that the relationship must be enforced to be a synchronization - otherwise a relation between specific states at particular processes is just a parallel state 2.5.

Since time is measured by clocks, we need to define what we mean by a clock:

**Definition 3.2.** A clock is anything that produces a monotonically increasing count of real numbers. We will distinguish between:

- **real time clocks**: in which the count is increased at constant time intervals.
- **event clocks**: in which the count is tied to specific events, which may follow each other at different intervals.

For example, a counter that is periodically incremented is a clock (in fact, the system clock used by Linux and Unix is a clock of this type). Clocks that measure actual time such as the system clock and standard human readable clocks (we sometimes call these "wall clocks" and refer to "wall clock time") are real time clocks. We can take the step number at each process as a kind of clock that
measures the progress of that particular process; this is an event clock in which the event that triggers the count is the transition between states at a process. (Note that in fact all clocks are really event clocks, since we are unable to produce exactly repeatable intervals to arbitrarily high precision. Every actual clock has some variation between the length of each tick). A direct consequence of this definition is that, if we have a deterministic serial (single) process 2.4, the time it takes is constant by the step number event clock (since it takes a repeatable number of steps), but not usually by the system clock (since repeated runs may take different wall clock times).

Given the above definitions, we now have:

**Theorem 3.3.** Let \( p \neq q \) be processes, \( c^p_i, c^q_j \) be the values of a clock \( c \) at each of the states \( s^p_i, s^q_j \) of those processes. The relations \( c^p_i = c^q_j \), \( c^p_i > c^q_j \) and \( c^p_i < c^q_j \) include all possibilities. In the first case, we say that the states \( s^p_i \) and \( s^q_j \) are simultaneous; in the second case we say that \( s^p_i \) is later than \( s^q_j \) and in the third case that \( s^p_i \) is earlier than \( s^q_j \).

**Proof.** Because the value of a clock is a single number, the relations \( =, >, < \) exhaust all possibilities. \( \square \)

**Corollary 3.4.** A synchronization 3.1 between a pair of processes enforces a relation between states that is described by one of these three relations >, <, = or one of the three combination of these relations obtained by the union of any pair: \( \leq \), \( \geq \), \( \neq \).

**Proof.** The relations >, < and = can be described as sets of pairs of real numbers. These relations over the real numbers (or the integers) are disjoint and universal - that is, given any pair of real numbers, one and only one of >, <, = holds. Therefore the intersection of any combination of these relations is empty and contains no states which can be enforced, and the union of all three is the entire universe so code to enforce this does nothing. This leaves the union of a pair of relations >, <, = which gives the three possibilities: \( \leq \), \( \geq \), \( \neq \). \( \square \)

(1) =equal; exemplified by a barrier or a synchronous message.
(2) \( \leq \)less than or equal (and its reverse: \( \geq \)); exemplified by a buffered message (sender \( \leq \) receiver).
(3) <less than (and its reverse \( > \)); exemplified by a message conveyed through a critical section in which the write action is separate from and earlier than the read action (sender < receiver).
(4) \( \neq \) not equal; exemplified by a mutual exclusion mechanism such as a semaphore or monitor.

A synchronization between more than two processes can then be defined by specifying relations between every pair of processes.

Although the number of possible combinations is enormous, in practice we generally restrict ourselves to simple combinations in which it is easy to specify (and code) the relation between all processes. Thus, a barrier is a synchronization in which times at all processes are =to each other, a mutual exclusion as enforced by a semaphore is a synchronization in which time at which all processes execute the protected code is \( \neq \) to time at whichever process holds the critical section.

3.1. **Synchronous transition.** We now introduce a deterministic parallel transition, which we will use in exploring deterministic execution.
Definition 3.5. Deterministic parallel transition: A parallel transition in which all processes advance.

\[ S_{G,J} \Rightarrow S_{H,K} \] 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, at least one process advances), and \( S_{G,J} \Rightarrow S_{H,K} \) is consistent with some pair of total states such that \( S_{\Gamma,J} \Rightarrow S_{\Gamma,K} \).

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 \( \rightarrow \) 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 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.

4. Dependences

We distinguish the following data-related dependences between statements in a program:

1. Direct dependence (Write-read): a variable \( X \) updated by statement \( s_1 \) is read at statement \( s_2 \), and \( s_2 \) appears after \( s_1 \) in program text.

2. Anti-dependence (Read-write): a variable \( X \) updated by statement \( s_2 \) is read at statement \( s_1 \), and \( s_2 \) appears after \( s_1 \) in program text.

3. Write-write dependence: a variable \( X \) updated by statement \( s_1 \) is also updated by statement \( s_2 \), and \( s_2 \) appears after \( s_1 \) in program text.

Each of these dependences constrain \( s_1 \) to execute before \( s_2 \) does, since a change in order of execution would mean, in case 1, that \( s_2 \) would get the wrong value of \( X \), and in cases 2 and 3 a change in order would leave the wrong value of \( X \) for any statements executing after \( s_2 \).

We can talk about direct dependences between communication statements at different programs:

1. Direct dependence (simultaneous of after): send followed by receive. A variable \( X \) read by send statement \( s_1 \) at process \( p \) holds data that is written into a variable \( Y \) by a read statement at process \( q \). Although the receive is logically after the send, if the send is blocking (as is the case for the default MPI mode), then the send may need to wait for the receive and the two statements are logically simultaneous.

2. Simultaneous: (direct) dependence: the same statement (e.g. a broadcast or a reduction statement) results in data being read from some variable \( X \) at some subset \( R \subseteq G \) and written to some variable \( Y \) at some subset \( W \subseteq G \). (For example, in a global reduction, data is read from a variable \( X \) at all processes in \( G \) and written to a variable \( Y \) at every process in \( G \).

We note that dependences carried by messages are always direct dependences, in which senders are required to execute before receivers. Collective operations look simultaneous because they involve multiple messages in their implementation that
must all complete for the statement to complete its execution at all participating processes.

We also identify a schedule dependence between a pair of communication statements at the same program; when a statement \( s_1 \) appears in the text before a different statement \( s_2 \). If \( s_1 \) is a blocking statement then its execution must complete before \( s_2 \) executes, even if there is no data dependence. If both \( s_1 \) and \( s_2 \) are blocking, then inverting the order of statements (allowed since there is no data dependence) leaves a schedule dependence in the same direction, since whichever statement appears first must execute before the other. Since the execution of a communication statement can depend on events at an external process, communication statements can carry dependences from one process to another; and schedule dependences between communication statements can spread a dependence across multiple processes even in the absence of true data dependences.

Consider three processes \( \{0, 1, 2\} \) logically arranged in a ring. Assume code as follows:

**Example 4.1.**

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>( s_1 )</td>
<td>recv ( X ) ( 3 \to 0 )</td>
<td>recv ( X ) ( 0 \to 1 )</td>
<td>recv ( X ) ( 1 \to 2 )</td>
</tr>
<tr>
<td>( s_2 )</td>
<td>( X = \text{func}(X) )</td>
<td>( X = \text{func}(X) )</td>
<td>( X = \text{func}(X) )</td>
</tr>
<tr>
<td>( s_3 )</td>
<td>send ( X ) ( 0 \to 1 )</td>
<td>send ( X ) ( 1 \to 2 )</td>
<td>send ( X ) ( 2 \to 0 )</td>
</tr>
</tbody>
</table>

This code is cyclic. We have direct dependences as follows:

- \( s_1^0 \to s_2^0 \to s_3^0 \to s_1^1 \to s_2^1 \to s_1^2 \to s_2^2 \to s_3^2 \to s_1^3 \) if the send is non-blocking;
- otherwise we have: \( s_1^0 \to s_2^0 \to s_3^0 \leftrightarrow s_1^1 \to s_2^1 \leftrightarrow s_3^1 \leftrightarrow s_1^2 \leftrightarrow s_2^2 \leftrightarrow s_3^2 \leftrightarrow s_1^3 \) where we use a double arrow to denote a send and receive each of which depends on the execution of the other statement.

We denote the process number by superscript, the statement number by subscript. Dependences from one process to another are direct send-receive data dependences, and at each process there are direct data dependences. Note that \( s_1^0 \) cannot execute until \( s_1^1 \) has already executed, and the same is true for every other statement. There is no way of breaking the cycle without an arbitrary decision as to which statement executes first to set a starting value for \( X \).

Consider instead the following code:

**Example 4.2.**

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>( s_1 )</td>
<td>recv ( Y ) ( 3 \to 0 )</td>
<td>recv ( Y ) ( 0 \to 1 )</td>
<td>recv ( Y ) ( 1 \to 2 )</td>
</tr>
<tr>
<td>( s_2 )</td>
<td>( A = \text{func}(X) )</td>
<td>( C = \text{func}(X) )</td>
<td>( C = \text{func}(X) )</td>
</tr>
<tr>
<td>( s_3 )</td>
<td>send ( A ) ( 0 \to 1 )</td>
<td>send ( B ) ( 1 \to 2 )</td>
<td>send ( C ) ( 2 \to 0 )</td>
</tr>
</tbody>
</table>

There is now no data dependence \( s_1 \to s_2 \) at any process, but there are still schedule dependences, which are described the same way as before: \( s_1^0 \to s_2^0 \to s_3^0 \leftrightarrow s_1^1 \to s_2^1 \leftrightarrow s_3^1 \leftrightarrow s_1^2 \leftrightarrow s_2^2 \leftrightarrow s_3^2 \leftrightarrow s_1^3 \)

Suppose we move \( s_1 \) so that it appears after \( s_3 \); this is allowed because there are no data dependences between \( s_1 \) and \( s_2 \) at the same process. We have:

**Example 4.3.**
This looks like it should work; with non-blocking sends we do not have a circular dependence. We have data send-receive dependences:
$s_3 \rightarrow s_1$ and $s_1 \rightarrow s_2$. We also have schedule dependences $s_3 \rightarrow s_4$ at each process. It appears that $s_3$ can execute at each process without depending on anything that happens at any other process. However, if sends are blocking we have: $s_0 \leftrightarrow s_1$, $s_1 \leftrightarrow s_2$ and $s_2 \leftrightarrow s_0$. That is, we have a forward data dependence, since $s_0$, for example, sends data to $s_1$, and a schedule dependence from $s_1$ to $s_0$ since a blocking send can’t complete before the receive executes. This gives us:
$s_0 \rightarrow s_0 \leftrightarrow s_1 \rightarrow s_1 \leftrightarrow s_2 \rightarrow s_2 \leftrightarrow s_3$; which is a cycle as before.

MPI non-blocking sends depend either on system buffers or programmer defined buffers; the code in example 4.3 is termed *unsafe* by the MPI standard, because it only works with sufficient resources. We can see this by considering what happens if execution is synchronous. Suppose statement $s_3$ executes at the same clock time at every process. Since statement $s_3$ executes (everywhere) after $s_3$, the data that was sent has to be somewhere (in the network? in a buffer?) between the execution of these two statements.

### 4.1. Rules for dependences

Dependencies are between statements (not processes - cross process dependences appear due to matching communication or synchronization statements).

**Direct dependences** appear between:

1. A write to a variable $X$ ($X = \text{something}; \text{CIN} \gg X$, $\text{PUT}(X)$) followed by a read from $X$ ($\text{something} = \text{function}(X)$; $\text{COUT} \ll X$. $\text{GET}(X)$).

   $\text{WRITE} \rightarrow \text{READ}$

2. A send of $X$ that matches a receive of $Y$. $\text{SEND} \rightarrow \text{RECEIVE}$. (It is possible that $Y$ has the same name as $X$; we label them differently here because they are in fact different memory locations at different processes.)

Although *send* reads a variable and *receive* updates, taken together the *send* can be considered to write (or *PUT*) a value to the receiver, which reads (or *GETs*) it.

Direct data dependences are also matched by schedule dependences in the same direction, since the sender/producer/writer of data must do its part logically before the receiver.

**Other data dependences** appear only inside the same process:

1. A *write*($X$) followed by another *write*($X$). (Subsequent reads will be incorrect if the order is reversed).

2. A *read*($X$) followed by a *write*($X$). (This read will be incorrect if the order is reversed).

In each of these cases there is a schedule dependence from the first statement to the second statement.

**Schedule dependences** are not usually considered dependences because they can (sometimes) be removed by changing the order of statements. We consider them because deadlock is frequently introduced by schedule dependences:
(1) There is a one-way schedule dependence $s_1 \rightarrow s_2$ if $s_2$ follows $s_1$ in program execution. This dependence may be reversed by switching the order of statements, but this is permissible only if there are no data dependences between $s_2$ and $s_1$. In the case of a loop, the dependence may be between different iterations of the loop and it may be impossible to switch the order.

(2) There is a one-way schedule dependence $s^p_1 \rightarrow s^q_2$ if statement $s_1$ at process $p$ must execute in order for a statement $s_2$ at process $q$ to execute. For example, a buffered send, where $s_2$ will wait until the data has been placed in the buffer, or an ordering synchronization that requires that $q$ wait for $p$ but does not block $p$.

(3) There is a two-way schedule dependence $s^p_1 \leftrightarrow s^q_2$ if statements at processes $p$ and $q$ must execute together. For example, if each statement needs confirmation that the other statement has executed before continuing, as in a synchronous send or barrier.

Schedule dependences of type 1 mean that the success of any statement in the program depends on the success of every previously executed statement. This in turn means that we need to be able to guarantee the success of every previous communication statement. This is complicated by the fact that to do this we in general need a dependence graph for the entire set of communication statements in the program.

One way of simplifying our analysis is to always write communications as complete sets of sends and receives, and put them into a function so sends and receives corresponding to one communication are not interleaved with sends and receives of another. (For example, an MPI reduction statement takes this form - it represents a set of sends, receives and computations, all executed as a single function call). In this way, we can analyze the communication inside the function as a separate dependence graph, without having to consider the possibility of a communication inside the function conflicting with one outside.

5. More to come

Notes to be added on:
- relation between determinism and absence of deadlock
- relation between proofs of determinism and deadlock freedom, practical constraints for execution.
- execution types and proofs:
  - trivially parallel execution
  - BSP execution
  - synchronous execution
  - MIPS execution