Parallel execution: CS624 notes

Ernesto Gomez
Contents

List of Figures 5

Chapter 1. Distributed and parallel computing 6
  1.1. Distributed and Parallel execution 6
  1.2. Goals of distributed execution 7
  1.3. Models of distributed execution: 7
  1.4. SPMD - a practical execution model. 11
  1.5. Other models: 11
  1.6. Performance model 12

Chapter 2. Definitions 15
  2.1. Hardware model 15
  2.2. Process model 16
  2.3. SPMD execution model 16
  2.4. Control flow graph 17

Chapter 3. States and transitions 28
  3.1. Serial state and execution 28
  3.2. Parallel state 29
  3.3. Parallel transition 31
  3.4. Parallel execution 33
  3.5. Determinism in parallel execution 37

Chapter 4. Synchronization and determinism 38
  4.1. Synchronous transition 39
  4.2. Communications 40

Chapter 5. Point to point messages 41
  5.1. Messages and synchronization 41
  5.2. A standard send-receive semantics 48
  5.3. Non-blocking messaging. 49

Chapter 6. Collective communication and complexity 51
  6.1. Point-to-point messages are sufficient 51
  6.2. Broadcast 52
  6.3. Reduction 53
  6.4. Semantics, and why we care about it 53
  6.5. Implementation of collectives: allreduce in MPICH and PC 54

Chapter 7. Dependences 55

Chapter 8. Networks: 59
## Contents

8.1. Definitions 59  
8.2. Network properties: 60  
8.3. Specific networks: 60  
8.4. Distributed systems - typical networks 62  

Index 64  
Bibliography 66
List of Figures

2.4.1 Skeleton Fortran program with basic blocks identified. 19
2.4.2 CFG for program in 2.4.1. Back edges are curved lines; natural loops are marked by dotted lines. 23
2.4.3 A compound natural loop 25
2.4.4 Non-reducible CFG. 26
2.4.5 Non-reducible graph transformed to reducible by replicating cycle. 27

5.1.1 Blocking send/recv 45
5.1.2 Non-blocking sends 46
5.1.3 Non-blocking receives 46
5.1.4 Non-blocking all 47
5.1.5 Deadlock resolved by schedule change 47
5.1.6 Deadlock freedom with non-blocking receive 48
CHAPTER 1

Distributed and parallel computing

A distributed computer system differs from a general parallel system in that we propose a specific architectural and execution model. Our architecture consists of a set of individual computers connected through a network, which leads to an execution of separate concurrent processes, each with its own memory resources.

In this course our focus is how to write programs for distributed computer systems. We will consider this from both a practical and a theoretical viewpoint. We will focus on concerns that are either unique to distributed programming or take on enhanced importance, when compared with standard (sequential) programming. We note two such issues in particular:

(1) The prevention or avoidance of deadlock.
(2) Deterministic execution.

The first can only occur in a parallel or distributed execution, the second is greatly exacerbated by parallel or distributed execution.

1.1. Distributed and Parallel execution

Main difference is due to hardware - distributed execution happens on a set of separate computers connected via network. Parallel execution is more general - it includes execution on a networked cluster, but also on a special purpose parallel supercomputer. A key feature of parallel hardware is that multiple processors are closely coupled, usually through hardware accessible by all CPUs, such as a shared memory.

Parallel supercomputers with specialized hardware have been built since the 1980's, using a number of specialized architectures. Some examples:

- Connection machine - up to 64000 simple processors, connected in a hypercube (see notes on networks) with a common clock and memory.
- KSR - an architecture in which all memory is cache, data is moved from one processor to another by remapping the cache. Many other specialized machines built.

Problems:

- None of the specialized parallel machines were good general purpose. Each was faster for a class of problems, but not good at all things.
- Since there is a small market for specialized supercomputers, the hardware can’t keep up in speed and capability with chips on commodity machines for which many millions are sold yearly. Wait 5 years, and the specialized parallel hardware is slower than cheap microcomputers.
As a consequence, supercomputer makers went bankrupt (Cray, KSR, builders of connection machine, many others) or stopped making them (Intel), leaving supercomputer users with software that will not run well or sometimes at all on any other machine.

The end result: all current supercomputers are in fact clusters of networked machines, with optimized networks and frequently in a fancy box. This is not because computing on a cluster is ideal - it is because when your supercomputer vendor goes broke, you can always build your own cluster from standard components and keep running your software. As a result, parallel and distributed computing have merged in fact into distributed computing.

A current example: Cray XD1 - uses AMD chips, SUSE Linux and a (highly optimized) ethernet connection, packed into a box.

Note that what we describe here is the 2008 state of the art: as hardware changes, computing will change also. A modern (past 2010) trend involves GPUs (Graphic Processing Units). These are the fastest floating point processors available today - a single top end card from ATI or NVIDIA has > 1 Teraflop performance, about 100 times faster than the fastest general purpose CPU. As a result, many supercomputer installations now consist of a small number of compute nodes fitted with as many GPU cards as possible. Because the parallelism in a GPU can be on the order of ~2000 (this refers to processing units), potentially a single GPU card has more compute power than a 1000 node compute cluster without GPUs. A GPU implements a less general computational model than a cluster of CPUs, so we are currently (circa 2015) computing on clusters of multi-core computational nodes, with each node having 2 or 3 GPU cards.

1.2. Goals of distributed execution

- Performance improvement:
- Address problems too big for a single computer
- Address a very large volume of small problems
- Respond to multiple realtime inputs faster than a single computer
- Fault tolerance

1.3. Models of distributed execution:

1.3.1. Type of parallelism: Two basic approaches to parallelism/parallel tools.

Implicit/automatic: Idea is to write standard sequential code, let the compiler/runtime system take care of parallelism.

Automatic parallelism at the instruction level is done by all modern processors via multiple instruction issue. Intel chips generally do up to 6 instructions at the same time, AMD up to 7. CPU dynamically analyzes instructions for dependencies
between them, reorders as needed, issues set of instructions with no dependencies
together. Average is less than maximum width of instruction issue, depends on
specific code being run. Rule of thumb - practical experience says this approach
usually gives parallelism of 5 or a little more. This works and is very useful, but
does not scale up to more than a few parallel processes.

At the language level, HPF (High Performance Fortran, developed at Rice
U.) was major effort. In practice, HPF needed extensions to standard Fortran to
express parallelism, but still avoided any reference to specific processes, and allowed
programmers to hint where parallelism might occur, leaving actual expression of
parallelism to the compiler. Programming HPF is not exactly like .
HPF achieved limited commercial and academic success, but was not asefficient
(and in some cases was harder to use) than low level message passing. It is not
currently popular.

Explicit parallelism: Programmer must be aware of processes, splits up the
work done by each process. Message passing and most shared memory systems
work in this way - data transfer and any synchronization code is explicit and must
be written by the programmer. Is a different programming paradigm, requires
different way of thinking - much like going from imperative programming to object
oriented.

As of 2008, the dominant paradigm is explicit parallelism, and most tools and
systems are designed for this.

1.3.2. Taxonomy and definitions: We need to distinguish between two lev-
els:

Logical (virtual, semantic) : What the system looks like to the programmer
and to programs running on it. For example, on a typical modern system, memory
management and the operating system make each program look like it is running
alone on the machine and has access to many gigabytes of RAM. The reality is that
multiple programs are running through time slicing, and the real memory is much
less than what the virtual memory system provides.

Real (actual, hardware): What the system really is. For example it is
possible to emulate a single large memory address space on a cluster (logical) but the
reality is multiple chunks of memory, each with its own address space on individual
nodes of the cluster.

"There Aint No Such Thing As A Free Lunch" - Robert Heinlein, The Moon
Is a Harsh Mistress is where I first ran into this, but it is probably earlier.
The idea here is that nothing comes for free - in our case we take it to mean
that anything we do to make the logical system act differently from the real system
is going to take effort, cost time and reduce efficiency. This is a consequence of
conservation of energy.

Rule of thumb - the greater the difference between real and logical system, the
slower the execution (on equivalent hardware).

1.3.2.1. Efficiency: An important consideration, since we are trying to solve
larger problems faster than on a single computer, is the effectiveness with which we
do this. Informally, we define the concept of efficiency:

\[
\text{Efficiency} = \frac{\text{Time to solve in parallel}}{\text{Number of processors}} \cdot \frac{\text{Time to solve on single processor}}
\]

Ideally we assume that, on \(N\) processors, we could solve the same problem \(N\)
times faster than on a single processor. An efficiency of 1 reflects this ideal case.
Typically there is some overhead involved in parallel or distributed execution (for example message delays), and the work to solve cannot be perfectly divided. This makes efficiency in most cases be less than 1, often a lot less.

Efficiency >1 is called superscalar; in general this is achievable when the parallel computation is done on a system that has substantially more resources than is possible on a single system. For example, total memory on a cluster may be such that the problem, once the work is distributed, can reside completely in RAM, whereas on a single computer much of the data must be swapped out to disk and incur much higher access cost as a result.

An equivalent concept is speedup:

\[
\text{Speedup} = \frac{\text{Time on single processor}}{\text{Time on N processors}}
\]

It is easy to see that a speedup of \(N\) implies an efficiency of 1, and vice-versa.

1.3.3. Best parallel algorithm and superscalar speedup. It is arguable whether superscalar speedup is achievable for algorithmic reasons; i.e. is there an algorithm which runs better in parallel (that is, does less work to solve the same problem) than the best sequential algorithm?

Assume that, for a particular problem \(P\), the best sequential algorithm is \(A\) and the best parallel algorithm is \(B\). Assume that \(B\) achieves superscalar speedup for algorithmic reasons. For this to happen, \(B\) must do less work than \(A\) to solve the same problem.

On \(N\) processors, \(B\) must be divisible into segments \(B_1 \ldots B_N\), where \(B_1\) runs on processor 1, \(B_2\) on processor 2 and so on. These segments must be independent of each other so that they can run concurrently; therefore they must be executable in any order. In particular, it must be possible to run them in the order \(B_1 + B_2 + \ldots + B_N\). But this ordering (or in fact any other ordering in which a single \(B_i\) runs at any given time) gives us a sequential algorithm, suitable for execution on a single processor. Call this algorithm \(A'\). Then \(A'\) does the same amount of work as \(B\), and therefore gives a sequential algorithm better than \(A\) and as good as \(B\).

That is, if we find a parallel algorithm superior to the best (known) sequential algorithm, we can use it to construct a new sequential algorithm that is just as good. Therefore we can not achieve superscalar speedup for algorithmic reasons.

The outline proof in the above paragraphs is incomplete in that, to really achieve the specified re-ordering of all segments \(B_i\) we are implicitly assuming no communications. Assuming a fixed number of communication statements in each concurrent segment, we really need to divide into sub-segments that end at the communication statements so that we can arrange receivers that immediately follow the corresponding senders, for example. So the proof is not as simple as here described. Nevertheless, the conclusion that superscalar speedup can not happen for algorithmic reasons is generally accepted.

Some of us (Segre, "Nagging"; Gomez and Scott, "Shortcutting" - references forthcoming) dispute this conclusion. We note that information is available in a different sequence in a concurrent execution that in a sequential execution. If we use this fact to guide the execution, we can obtain an execution schedule and path to solution that cannot be computed in a sequential execution (although it can be followed once the path is known). We may discuss this in more detail later in this class, but it is a controversial research topic in which we are distinctly in a small minority.
1.3. MODELS OF DISTRIBUTED EXECUTION:

1.3.4. Classification by Type of communication:

1.3.4.1. Shared memory (one-sided access): Also referred to as PUT/GET semantics. Processes running concurrently have a single memory address space, and all can read from anywhere in the memory (GET) or write to anywhere (PUT). We call this one-sided access because a process can do a PUT or a GET on its own, without need for interaction with (or permission from) any other process. PUT and GET do not have to appear explicitly; \[ x=y \] means GET value of \( y \), PUT into \( x \).

On a single computer, memory access has the Random Access Property: that is, it takes essentially the same amount of time to get data from any location in memory. This frequently also holds for supercomputers with shared memory. On a cluster, we can simulate shared memory, but we generally get what is called \textbf{NUMA (Non-Uniform Memory Access)} rather than true random access - time to get data varies according to where it is in memory, and is slower if the process is running on one node and the physical memory is on a different node.

1.3.4.2. Message Passing (two-sided communications): SEND/RECEIVE semantics. Each process has its own address space. To get data from one process to another, the source process executes a SEND instruction and the target process executes a matching receive instruction. We call this two-sided because both a sender and receiver process must execute specific actions for the data transfer to happen. Notice that two-sided is a lower bound; message passing can involve multiple processes; for example we may have a single sender and many receivers for a broadcast.

The actual syntax may not use SEND and RECEIVE instructions; for example in the Planguage (see textbook), \( x@2=y@1 \) means send the value of \( y \) from process 1 to receive it into the variable \( x \) at process 2. This looks a lot like \( x=y \), which we called one-sided access, but notice that it explicitly mentions two different processes, each of which takes a different action. This statement in fact means:

\[
\begin{align*}
& \text{IF( } ME==1 ) \text{ send } y \text{ to } 2 \\
& \text{ELSIF( } ME==2) \text{ receive } x \text{ from } 1 \\
& \text{ELSE ignore.}
\end{align*}
\]

Both one-sided and two-sided access can be used in either the logical or real layers. For example, we can emulate one-sided access on a cluster with remote procedure calls: A process \( P \) running on node \( A \) can send a message that invokes an RPC on node \( B \) that in turn sends back a message with data. To \( P \) this looks like a GET, but in reality it is a message transaction. \( P \) can ignore the fact that it is in reality communicating with a process at \( B \) because that process is a daemon preset to respond to certain messages at any time, so \( P \) does not have to take it into account. Alternately, we can set up messages between processes \( P \) and \( Q \) running on a shared memory machine, using two shared memory locations. Call these locations \( Flag \) and \( Data \). To send from \( P \) to \( Q \), process \( P \) first writes the information to \( Data \) and then sets \( Flag=TRUE \). Process \( Q \) periodically checks \( Flag \); when it changes to \( TRUE \) it reads the data and resets \( Flag \) to prepare for more transmissions.

Most real systems are in fact mixed. Shared memory systems (either logical or real) still need synchronization (such as the shared flag in the example above) to ensure that data is actually transferred; synchronization is a two-sided action because it affects all processes involved, and must be executed by all of them. Also, each process in a shared memory system usually will also have local data which is not shared.
Processes in message passing systems generally only have local data which is not shared, but they still need some form of one-sided access to set up message send/receive protocols.

1.3.4.3. Flynn’s taxonomy: (information flow, used for hardware/exec models).
Classification by differences in instructions and data streams. A single instruction stream means the same sequence of instructions runs in every case. A single data stream means that all processes see the same data in the same sequence. We have 4 overall possibilities:

<table>
<thead>
<tr>
<th>DataStream</th>
<th>InstructionStream</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single</td>
<td>SISD</td>
</tr>
<tr>
<td>Multiple</td>
<td>SIMD</td>
</tr>
<tr>
<td>Multiple</td>
<td>MISD</td>
</tr>
<tr>
<td>Multiple</td>
<td>MIMD</td>
</tr>
</tbody>
</table>

- SISD is standard sequential execution - a single instruction stream on a single data stream.
- SIMD is data parallelism - run the same instruction sequence at each process, but have different data.
- MISD is run different instructions on the same data. This applies, among other things, to backtracking algorithms in which we do different things or follow different paths to potential solutions on the same data set.
- MIMD is the most general - it contemplates running different instruction streams on different data at each process. If you have hardware capable of MIMD, you can run in any of the other modes. All clusters of computers are capable of MIMD, however some specialized supercomputers were designed for a more limited execution model, usually SIMD.

1.4. SPMD - a practical execution model.
This stands for single program - multiple data. Sounds a lot like SIMD, but notice that all we are saying is that we have multiple copies of the same program running concurrently, but not that each program is doing the same thing to the data. Also known as procedural or task parallelism - we start out with the same code at all processes, but they are allowed to do different tasks. This is really a specialized form of MIMD. The advantage is that it is easier to write a concurrent program, and coordinate a concurrent execution, if we can make the assumption that all processes have code in common. Most real world parallel programming systems are SPMD, or at least support it, and it is the most common paradigm for concurrent programs designed to split up work to solve large problems.

Specifically, Plaguages are SPMD, and MPI programming is mostly SPMD (although MPI actually supports a more general model).

1.5. Other models:

1.5.1. Theoretical models. PRAM - Parallel Random Access Machine. Used to calculate upper bound on complexity of parallel computation. Extends RAM model (essentially a Turing Machine with random access memory of arbitrary size instead of tape) by adding multiple processors. Assumes number of processors can increase as a function of problem size, assumes all processors are exactly synchronized, and any processor can access any memory with fixed (usually one cycle delay).

Circuit - Computation is modeled by a circuit. Equivalent in compute power and speed to PRAM.
1.6. PERFORMANCE MODEL

**CSP** - Communicating Sequential Processes - C.A.R. Hoare. A theoretical model in which individual sequential processes execute concurrently, and communicate at a rendezvous where 2 processes execute specific (synchronous) communication code. Important because CSP has been implemented as a real language, serves as practical and theoretical justification for the statement that synchronous message passing is sufficient for any parallel computation.

**BSP** - Bulk Sequential Parallelism. See http://www.bsp-worldwide.org/

1.6. Performance model

**Amdahl’s Law:**

Amdahl’s Law is an attempt to predict how much speedup can reasonably be expected in parallel and distributed computation. We know that a very few problems are trivially parallel - they can be split up into many totally independent tasks. Such problems will allow arbitrary amounts of speedup by using more processors. For example, SETI at Home uses several million machines to get a speedup of several million compared to any single machine, because all computations are essentially independent. (Of course, how many processors we can use also depends on the amount of available data. If we have, say, 1 terabyte of data for a trivially parallel problem, we still won’t be able to get a speedup of 1 trillion (10 to the 12th for American trillion) because we could only distribute one byte to each processor, presumably not enough to do any useful work.

Amdahl notes that any program for any algorithm that is not trivially parallel involves multiple stages which do different things and offer different possibilities for parallelism, or even none at all. For example, each merge stage in merge sort offers half the potential parallelism as the previous stage (consider - to merge 16 files we can use 8 separate processes, giving 8 files which allows 4 processes). Frequently initialization stages and final output need to be done by a single process, and in many cases there are intermediate stages which are sequential as well. For purposes of setting an upper bound, Amdahl assumes that the total work \( W_t \) done by a program can be divided into two main components, which we will call \( W_s \) for work that must be done sequentially by a single process, and \( W_p \) denoting work that may be parallelized.

\[
W_t = W_s + W_p + \text{Overhead}
\]

We take work in the above equation to be number of steps, in the same way we do when estimating complexity (big O). Overhead is any extra work we need to do to run in parallel, we assume it is 0 in the sequential case. Main difference is, when estimating work, we do not throw out lower order terms and constants as we do in complexity calculations. Since work thus estimated is total number of steps needed to solve a problem, we can just multiply it by some constant (steps per second, for instance) to get an estimate of the runtime on a particular computer so we can use work as a stand-in for time in speedup and efficiency calculations. For convenience we can take the constant to be 1, and then we can use time and number of steps interchangeably; following our notation above we can talk about Ts as the time of the sequential portion of the program, andTp as the time of the part of the program that may be parallelized. Also for convenience, and since we want an upper bound, take the best case and assume overhead is 0. Let \( T_N \) be the time on N processors, so \( T_1 \) would be sequential execution time. It follows that:

\[
T_N = T_s + \left( \frac{Tp}{N} \right)
\] and \( T_1 = T_s + Tp \)
Therefore:

\[ Speedup = \frac{T_1 = T_s + T_p}{T_N = T_s + \left(\frac{T_p}{N}\right) } \]

Assume that the number of processors becomes arbitrarily large, the term \( \frac{T_p}{N} \) is arbitrarily close to 0 and may be ignored.

Therefore, the maximum speedup obtainable even with an effectively infinite number of processors working in parallel is:

\[ Speedup = \frac{T_s + T_p}{T_s} = 1 + \left(\frac{T_p}{T_s}\right) = 1 + \left(\frac{W_p}{W_s}\right) \]

That is, since we can’t parallelize \( W_s \) at all, and must always do it, the maximum possible speedup just depends on what fraction of the work can be done in parallel (\( W_p/W_s \)).

We now estimate what this fraction (\( W_p/W_s \)) is in a typical case. Of course, it depends on the problem, but it seems not unreasonable to assume that most of the time (\( W_p/W_s \)) would have a value of 10 to 20, which then sets an upper limit on the parallel speedup we can expect. It therefore would seem pointless to have a parallel computer with much more than 10 to 20 processors, since even with hundreds or thousands of machines we are still not going to get more than a tiny fractional improvement over 10 processors.

What’s wrong with this picture?

Experience with real-world problems shows that, whereas our guess of limited speedup holds for small problems (indeed, we often achieve substantial parallel slowdown, perhaps due to the overhead term that we ignored), it is often the case for large problems that we can get speedups of several hundred or thousand with that many processors. IBM is currently designing and building supercomputers using what they call the Blue Gene architecture (many processors on a chip, many chips on a board, fast network connections), ultimately intended to produce a supercomputer with a million processors, betting that such a machine would yield a speedup of hundreds of thousands and make practical the solution of large genetic and molecular dynamics problems.

There is a hidden assumption in Amdahl’s Law. Go back to our work estimate:

\[ W_t = W_s + W_p + Overhead \]

\( W_s \) and \( W_p \) are estimates for different segments of the program, involving different code and different algorithms. A naive interpretation of Amdahl’s Law assumes that the fraction (\( W_p/W_s \)) is constant, but this is only true if the complexities are the same, that is if \( O(W_p) = O(W_s) \). There is no reason to expect this equality, and in fact we would in general expect different complexities. If, for example, \( O(W_s) < O(W_p) \), then as problem size increases, so does \( W_p/W_s \) and we would expect greater speedup. In this case, we say that the algorithm is scalable; that is parallelism is more efficient for large problems. Of course, it is also possible to have \( O(W_p) = O(W_s) \) in which case our original interpretation of Amdahl’s Law holds, or even \( O(W_p) > O(W_s) \), in which case we get less parallelism for large problems. We say such algorithms or programs are not scalable.

Note that scalability can be a property of an algorithm or of a problem (just like complexity). The same problem may have several different algorithms of different complexity and scalability. Just as in complexity, we can sometimes talk about problems that are scalable or not - for instance, all trivially parallel problems are scalable, there are some inherently sequential problems that are not scalable (if the problem is not scalable, we are saying it has no good parallel algorithm). Also note that the best sequential algorithms are not necessarily the most scalable - for
example, shell sort, an excellent $O(n \log(n))$ algorithm is not particularly scalable, whereas the much-maligned bubble sort scales well.

In practice, scalability also requires consideration of the overhead term. Whereas $W_s$ and $W_p$ are in general functions of data size and increase as time complexity, Overhead is generally a function of number of processors. We can estimate communications complexity in terms of communication patterns and number of processors, and find that it is usually at least $\log(n)$ in the number of processors and often worse. This leads to cases in which a program runs more efficiently for larger data sizes and number of processors, but only up to some limit after which communication complexity brings efficiency down again. Many programs find their highest efficiency with several hundred or low thousands processors. How to deal with communications on more than tens of thousands of processors is a research problem at this time (2008).
CHAPTER 2

Definitions

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.

2.1. Hardware model

Definition 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.

### 2.2. Process model

**Definition 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 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).

### 2.3. SPMD execution model

**Definition 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.

### 2.4. Control flow graph

The control flow graph (CFG) is most frequently defined after the program has been transformed to some intermediate code representation, usually 3 address code [AHO 85]. Translation into this intermediate code loses structural information (for example N-way branches) that we will be able to use in analysis and in implementing efficient run time support for identifying implicit process groups. We will here extend the definition so we may apply it to more general analysis, including higher level code, specifically code which may be in languages such as Fortran or C. In order to do this we will need to extend the standard definition of basic block to take into account the richer control flow constructs available in higher level languages, and use this extended definition in our CFG.

First we must define what we mean by a block. We can identify blocks of code in a program as a set of statements that are executed consecutively. Examples of this are the statements in the scope of an IF statement or loop. Blocks are frequently designated by textual block delimiters such as matching brackets \{ \} in a C program, or matching THEN, ELSE and ENDIF statements in Fortran. Such delimiters are defined as part of the syntax of a language, they are not always required.

**Definition 4.** A **code block** is a set of contiguous statements in a program such that control may only reach a statement in the block by passing through the first statement.

A subroutine or function is always a code block; even though it may be called from multiple points in the program, control normally enters at the first statement (there are exceptions to this in Fortran which we will discuss below). It is possible, however for code blocks to be nested inside other code blocks. For example, the body of a natural loop is a code block. It may contain an IF-THEN-ELSE-ENDIF
statement, which delimits two code blocks: one which executes if the predicate of the \texttt{IF} statement evaluates to \textsc{true}, and the other if it is \textsc{false}.

To define a control flow graph (CFG) we will indicate control transfers from one block to another with arcs, but since code blocks may be nested, they cannot serve as nodes. We will define a \textit{basic code block}, or simply \textit{basic block}, as the largest group of statements that we could graph as a single node. The following definition extends the standard definition of [AHO 85] by adding communication statements and block delimiters to the definition of leader.

\textbf{Definition 5.} A \textbf{basic block} is a set of executable statements in a program that must always be executed from start to finish or not at all. Control may only enter at the first statement, called the \textit{leader}, of the basic block, and if execution reaches the first statement, then it must continue to the last statement of the block.

Leaders are identified as follows (based on Aho & Ullman):

i. The first executable statement of the program is a leader.

ii. Any executable statement which is the target of a (conditional or unconditional) jump or control transfer is a leader.

iv. Any executable statement which immediately follows a conditional statement is a leader.

v. Any executable statement following a block delimiter is a leader.

vi. Any synchronization or communication statement is a leader.

(Intuitively, some data transfer occurs at a synchronization, which changes the data set at each process. We are going to define state of a process in terms of data in memory and basic block being executed; data received from another process implies a change of the serial state at each process that cannot be determined from data at that process.)

The body of a basic block is the set of statements in the text starting with a leader and up to but not including the next leader, or up to the end of the program text if no leader is encountered.

Note that the above definition implies that there can be only one communication statement in a basic block; we start a new block whenever we encounter such a statement. The reasoning behind this is that communication statements can change the data set unpredictably at the communicating process. We take something that changes the state in some way that is not determined by the data present in the process at entry to the basic block as changing also the basic block. The same reasoning applies to synchronization statements, which might involve some data or information transfer. The key point in the basic block is that its execution is totally predictable from the contents of memory on entry to the block and the position of the block in the CFG. We are only concerned with executable statements for the purpose of defining basic blocks, but we include (as executable) statements such as \texttt{continue} in Fortran which specify no operation, but may be targets of conditional or unconditional jumps, or indicate the range of a loop.

Logic statements in Fortran or C include added text to delimit the range of the statement. For example, the \texttt{IF} statement may be followed by \texttt{else}, and \texttt{endif} in Fortran, or by a block of code enclosed in brackets in C. The \texttt{IF} statement itself follows the rules for conditionals and ends the preceding block; the others are delimiters; they should not be considered executables, but they do define blocks of code.
Figure 2.4.1. Skeleton Fortran program with basic blocks identified. Direct analysis of high level code creates a different block structure than analysis of intermediate code. For example, block 2 would translate (on most hardware) to a pair of conditional jumps and thus be split into two blocks. 

```fortran
program foo
    statement set A
    broadcast
    if x1 goto 10, 20, 30
    10 continue
    statement set B
    20 continue
    do i=.. statement set C
    if x2 then statement set D
    Pstream reduce
    else
    do k=.. statement set E
    Pstream reduce
    end do
    endif
    end do statement set F
    Pstream broadcast
    Global reduce
    statement set G
    goto 90
    30 continue
    do j=.. statement set H
    Pstream broadcast
    if x3 goto 40
    40 continue
    Global reduce
    Send-receive
    statement set I
    90 continue
end
```
Loops in higher level languages require special consideration, due to the complexity of control structures. In general, the first statement in a loop is a leader, and the last statement in the loop ends the final basic block. Statements such as do{... } in C should be considered textual delimiters of a block of code rather than executables. However, statements such as while(), until() and for() are logic statements and end a basic block. If such are at the end of a loop (e.g. do{...}while(); ) they may be considered part of the final block in the loop, since execution of that final block always continues through the statement in question.

Loop control statements at the beginning of a loop are basic blocks unto themselves. Since they may prevent execution of the whole loop, they cannot be part of the first block inside the loop. Since control passes from inside the loop back to the first statement to determine if the next iteration should be executed, this statement cannot be part of the block just before the loop. Loop control statements may be at the same time targets of jumps and conditionals, so they meet the requirements for start and end of a basic block.

The Fortran DO loop statement may apply up to the next END DO, which may be considered a textual marker (like brackets in C). There are alternate methods of specifying the scope of a loop in Fortran, such as ending on a numbered statement. We will here limit ourselves to consideration of loops which could be written with DO .. END DO, and avoid complications that may arise with older versions of Fortran which allow jumps into the body of a loop, or to its last statement.

It is possible in Fortran to have a set of nested loops which each use the same numbered statement as the last statement in the loop. If the statement performs no operation, as in a CONTINUE statement, then a single label can be replaced, for purposes of analysis, with a set of END DO statements. If the statement is an executable operation, we should consider it to belong to the innermost loop, which, being executed by all the outer loops, makes the statement execute inside all loops.

Definition 6. A program control flow graph, or CFG, is a directed graph: $F = \{V, A, s, E\}$,

in which the vertices $V$ are the basic blocks in the program and the arcs $A$ denote control transfer from the end of one basic block to the beginning of another. $E$ is the set of end nodes, in which the program may halt and the node $s \in V$ is the first basic block in the program, where execution always begins.

We have added a start node and a set of end nodes to the conventional definition of CFG because we will require them in our analysis. We do not believe that this imposes a serious restriction on the set of programs that can be thus analyzed. While it is possible to imagine a program, such as an operating system, which is supposed to loop for an indefinite period, it should not be seriously restrictive to allow the possibility of a halt instruction which would send it to an end node (such an instruction could be added if not already present).

A subroutine or function call may be represented as a vertex in the CFG. Even though a subroutine may have multiple exit points in the code, it always returns control to the statement immediately following the call, so the call looks to the external program like a basic block. The code of the subroutine itself is then represented by a separate CFG.

It may be imagined that a program could be started at several different nodes, perhaps by using several different program launch routines (note that this is not the case of a program which starts at a single node $s$ and immediately jumps to
one of a set of other nodes). It is possible that the same program text started in different places will result in drastically different execution. It is clear that the properties of the program could be drastically different; for example some nodes could be reachable from one start point that are not reachable from another. We will therefore require a single start node, and if we somehow manage to launch the same program code from a different node we will consider this to be a different program.

(An exception to subroutines having single entry points is found in Fortran subroutines that use the entry statement. This allows defining different entry points, call parameters and return types within a single subroutine. The implications of multiple entry points in a subroutine are the same as for a program; i.e. the CFG for the subroutine will in general be different, have different connectivity and data dependences. Therefore a subroutine with multiple entries has to be treated as a different subroutine for each entry statement).

If we have a CFG that corresponds to a program with multiple end nodes we will need to transform this for purposes of our later analysis to a CFG with a single end point. This can always be done without changing the meaning of the program:

**Theorem 7.** Given a program \( P \), with a CFG \( F = \langle V, A, s, E \rangle \) in which \(|E| > 1\), we can transform \( P \) to a program \( P' \) with \( F' = \langle V', A', s, E' \rangle \) and \(|E'| = 1\); that is, a program with a single exit node, and such that \( P' \) and \( P \) perform the same computation.

**Proof.** Construct \( P' \) as follows: Add to \( P \) a single end node \( e \) such that \( E' = \{ e \} \), such that the only instruction in \( e \) is a stop or end statement. Replace any stop or end statements in every node in \( E \) with jumps to \( e \). Every node in \( E \) will then have an exiting arc to \( e \), and execution will follow that arc in \( P' \) whenever it would have halted in \( P \). \( P' \) executes the same code as \( P \) until program end, and then only executes one additional jump which does not modify any variable in memory or produce any output. Therefore computed results are unchanged. \( \square \)

Given the application of the same algorithm to generate the CFG, we can be sure that only one CFG corresponds to each program:

**2.4.1. Reducible CFG.** We now need to define what we mean by a reducible CFG, for which we will need several other definitions. In most of these we are following Aho & Ullman [AHO 85].

**Definition 8.** DOM relation. Given a CFG \( G = \langle V, A, s, E \rangle \), and nodes \( d, n \in A \), \( d \) dominates \( n \) (which may be written \( d \text{ DOM } n \)) if all paths \( s \rightarrow^* n \) are of the form

\[
s \rightarrow^* d \rightarrow^* n
\]

That is, all paths from \( s \) to \( n \) pass through \( d \).

Note that the DOM relation depends on the identification of a start node. Consider, for example, a loop containing several nodes. If we select any one node in the loop as the start node, it will dominate all the others (the start node always dominates all reachable nodes). It will be convenient in our development to select a particular node that is not the start node of the CFG as the start node for DOM. In such a case, our selected start node will dominate all and only the nodes that are reachable from it. [AHO 85] describes algorithms for efficiently computing DOM.
Some properties of DOM, proved in [AHO 85] are:

**Lemma 9.** \( \text{DOM is a reflexive partial order} \)

Also:

**Lemma 10.** \( \text{Dominators of a node } n \text{ are linearly ordered by } \text{DOM}. \)

Dominators may be displayed as a **DOM tree**, in which the start node \( s \) is the root node, the parent of each node is its immediate dominator, and all and only the dominators of a node \( n \) are its ancestors in the tree. The DOM relation may also be used to classify edges in a CFG:

**Definition 11.** A **back edge** of the CFG is an arc \( a \to b \) such that \( b \) (the head) dominates \( a \) (the tail).

That is, back edges follow paths that take us closer to the start node \( s \).

**Definition 12.** A **forward edge** of a CFG \( \langle V, A, s, E \rangle \) is an arc \( a \to b \) such that there is an acyclic path of the form \( s \to^* a \to b \).

Such an arc is a member of a set of edges which, taken together, form an acyclic graph in which every node can be reached from the start node \( s \) of the CFG, the definition in [AHO 85]. That is, if all edges were forward edges, the entire program would be an acyclic graph.

We now introduce a specific class of CFG.

**Definition 13.** A **reducible flow graph** is a CFG \( \langle V, A, s, E \rangle \) in which all edges in \( A \) can be partitioned into:

- **Forward edges**, with the property that the set of all forward edges forms a directed acyclic graph (DAG) in which all nodes are reachable from \( s \), and:
  - **Back edges**, of the form \( a \to b \) such that \( b \) DOM \( a \).

Note that the notion of reducibility is closely related to the presence of cycles in a CFG. An acyclic CFG is automatically reducible, since all its edges are forward edges. A CFG in which all loops have single entry points is also reducible, since such loops have graphs in which all nodes are dominated by the entry node. Therefore an edge that loops back to the entry node is a back edge - it is dominated by its head. However, a loop with more than one entry point will not have this property. Consider a loop with two entries: since nodes in it will be reachable on paths through either entry node, neither node dominates the loop. Therefore the edges that form the cycle by looping back to the entry nodes are not back edges, and they are not forward edges either because they form a cycle.

**2.4.2. Loops in reducible CFG.** We will initially limit our consideration to programs that can be represented by a reducible CFG. All edges in such a CFG are either forward or back edge, and loops are easily and unambiguously identified in such programs.

**Definition 14.** The **natural loop** \( L \) of a back edge \( x \to s \) is the set of those nodes that have paths to \( x \) without passing through \( s \), and \( s \) itself, called the **header** of the loop. ([AHO 85], among others, give algorithms for identifying natural loops). Note that \( s \) dominates every node in the loop; that is, there are no arcs to any node in \( L \) from any node outside \( L \) that do not pass through \( s \).

From [AHO 85] we have:
Figure 2.4.2. CFG for program in 2.4.1. Back edges are curved lines; natural loops are marked by dotted lines.

Lemma 15. In a reducible CFG, all cycles are natural loops.

We will be using the idea of an exit node from a loop in much of our continuing development, so we need to establish the following property:

Theorem 16. In a natural loop, an exit node has outdegree greater than one, and at least one immediate successor of an exit node is outside the loop.

Proof. Assume false. Let there be an exit node $y$, of outdegree greater than one, such that all paths $y \to z$ are such that $z$ is inside the loop. Let the back edge of the loop be $x \to s$. Then all immediate successors $z$ of $y$ must have a path $z \to x$ without passing through $s$ (by definition 14), and $y$ is not an exit of the loop. Assume instead that $y$ is an exit node of outdegree one, and has an immediate successor $y \to w$ such that $w$ is outside the loop. Then no successor of $y$ is in the loop, and neither is $y$ itself since it cannot reach $x$ without passing through $s$. □
2.4. CONTROL FLOW GRAPH

2.4.3. Loops with multiple exits. Let $L$ be the natural loop of the back edge $x \to s$ (def: 14). We know from the definition that we can not have any arc to any node in $L$ except $s$ from a node outside $L$, because $s$ dominates $L$. It is, however, possible to have a back edge $z \to s$, where $z \neq x$.

In this case we can define two natural loops with the same header. Let the natural loop of $x \to s$ be $L$, and the natural loop of $z \to s$ be $M$. Both have the same header, $s$. There are nodes $y \in L \cap M$ from which both $z$ and $x$ may be reached, and also possibly nodes that are in $L$ or $M$ but not in both. The definition of back edge is not violated, because $s$ dominates both $x$ and $y$; neither is the definition of natural loop violated.

We also see that the definition of reducible CFG (13) is not violated, because if we remove both back edges then what is left is a DAG, of only forward edges. What we have, in fact, is a loop with some common code and two different loop conditions which send execution back to $s$; this is one of the ways to have a loop with two exit points. We can obviously extend this discussion to a situation where we have three, four, or in general $N$ back edges all pointing to the same header $s$, giving us a loop with $N$ exits and one entry.

We will define:

Definition 17. A compound natural loop is either the natural loop of a single back edge, or the union of the natural loops of a set of back edges which all point to the same header node. We may simply call these compound loops, recalling however that a simple natural loop is included in the definition (see fig 2.4.3).

Compound loops may be identified by finding all the natural loops in a CFG and then taking the union of loops that share the header node. A property of a compound loop in a reducible CFG is that if execution enters the loop at the header node, it may only leave the loop through one of the exit nodes of the compound loop. Also, by the definition (14) of natural loop, it is not possible to enter the compound loop except at the header node. A process executing along any branch in the compound loop must return to the header node. Therefore it may select a different branch for each iteration, and exit on any of the possible exits.

Therefore a compound loop defines a subgraph within the CFG with a single entry point and $N$ exit points, each of which is a loop exit.

2.4.4. Non-reducible CFG. Studies of random samples of Fortran programs show that most such programs do have reducible flow graphs [ALLEN 70, KNUTH 71]. These studies show that even when programs are written using GOTO statements, in a language such as Fortran which is not designed for structured programming, programmers tend to write reasonably well structured code which leads to reducible CFG. Even so, we need to be able to deal with a non-reducible flow graph when it occurs.

A non-reducible CFG is characterized by a cycle in which there is at least one edge that is neither a back edge nor a forward edge. That is, its tail does not dominate its head, which would make it a forward edge, nor does its head dominate its tail which would make it a back edge. We find that typically this occurs when we have a cycle with more than one entry point. It is not a natural loop because there is no node identifiable as the head of the loop.
Consider a cycle with two entry points at different nodes. This can be transformed to a reducible graph by replicating all the nodes in the cycle and connecting one of the entry arcs to one set of nodes, and the other entry arc to the other. That is, convert one cycle with two entries into two duplicate cycles each with single entries. Having done this, the resulting pair of graphs is each a natural loop, and reducible.

Techniques for converting non-reducible CFG to reducible can be found in [AHO 85] and other compiler references.
Figure 2.4.4. Non-reducible CFG. A subgraph from CFG of Figure 2.4.2, with arc (3,7) added. Once this is done, arc (7,4) is no longer a back edge. Nodes 4,5,6,7 are now a cycle with two entry points (4 and 7), and no longer a natural loop.
Figure 2.4.5. Non-reducible graph transformed to reducible by replicating cycle.
States and transitions

3.1. Serial state and execution

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 18.** **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 \(i\),
- \(D_i\) is data input when block \(x_i\) is executed.

Take \(M_0\) to be the contents of memory at the start of block \(x_i\); then \(M_i = x_i(M_0, D_i)\). 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 19.** **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 \(x_i\); we will usually write the abbreviated form \(\sigma_i \rightarrow \sigma_{i+1} = (x_i, M_i) \rightarrow (x_{i+1}, M_{i+1})\). Note that the state of memory at the end of execution of block \(x_i\) is the same as the state of memory at the start of block \(x_{i+1}\); that is \(M_{i+1} = M_0(i+1)\).

Given the transition, we now define execution as a series of transitions:

**Definition 20.** **Serial execution:** \(e = \sigma_0 \rightarrow \sigma_1 \rightarrow \ldots \rightarrow \sigma_N\) for a non-divergent execution (that is, one which terminates) in \(N\) steps, where \(\sigma_0 = (s, M_0)\) and \(s\) is the start block. We will often designate the final state as \(\sigma_e = (x_e, M_e)\) 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 \(e = \sigma_s \rightarrow^* \sigma_e\) to denote 0 or more transitions between a start state and an end state. For a possibly divergent execution we will write \(e = \sigma_s \rightarrow^* \sigma_e |\perp\), where \(\perp\) 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 21.** 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 \( \sigma_i \) be a state. In serial execution there is a transition \( \sigma_i \rightarrow \sigma_{i+1} = (x_i, M_i) \rightarrow (x_{i+1}, M_{i+1}) \). 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 \( x_i \). Given that \( x_i \) contains no explicitly non-deterministic or random 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 \( \sigma_i \rightarrow \sigma_{i+1} \) has only one possible outcome and is deterministic.

Given a sequence of data inputs \( D \), an initial state \( \sigma_0 = (x_0, M_0) \) the transition out of \( \sigma_0 \) 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. \( \square \)

Take now a set of \( M + 1 \) processes \( \Gamma \) and a data set \( \Delta = \bigcup_{p=0}^{M} D_P \), where the superscript denotes the process number from 0 to \( M \). It is tempting to define a parallel execution as a set of serial executions \( E_{\Gamma, \Delta} = \{ e_P \} \). The problem with this approach is that it does not lead to any useful way of defining a parallel state.

### 3.2. Parallel state

We instead start by defining a parallel state as a collection of individual process states:

**Definition 22.** Parallel SPMD state:

\[
S_{G,J} = \{ \sigma^p_j | p \in G, j \in J, G \subseteq \Gamma \}
\]

\( G \) 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 \( G \). \( \Gamma \) is the ordered set of all processes. For an execution \( E_{\Gamma} \), if \( G = \Gamma \) then we call \( S_{\Gamma,J} \) 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.

#### 3.2.0.1. 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. We frequently find such process sets defined, 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
3.2. PARALLEL STATE

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 + Bugheri a uniform value is said to be \textit{global},
a non-uniform value is called \textit{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. 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 23. 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.

3.2.0.2. Functions of parallel state. We will need several functions of a parallel
state:
\[
\beta(S_{G,J}) = X_{G,J} = \{x^p_j \mid p \in G\}, \text{ that is, the ordered set of basic blocks which}
\text{the processes in } G \text{ are executing at step } j \in J; \text{ and}
\mu(S_{G,J}) = M_{G,J} = \{M^p_j \mid p \in G\}, \text{ that is, the ordered set of memory contents}
\text{of processes in } G \text{ at step } j \in J.
\]
These functions for a parallel state are equivalent to functions of a serial state
that select the basic block or the contents of memory.

3.2.0.3. Extension to definition of basic block. Take state \(S_{G,I}\) which is consistent
with a total state \(S_{\Gamma,I}\). Let \(M_{G,0I}\) 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
\(M_{\Gamma,0I}\) 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 18. There we had:

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

Now, since there may be communication between processes, we have that \( M^P_i \) is a function of not only \( M^P_{0i} \) 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 \( M_{\Gamma,1} \). We have:

\[ M^P_i = X_{\Gamma,1}(M_{\Gamma,0}, M \leq_t M_{\Gamma,1}, D_{\Gamma,1}) \]

where the \( M \leq_t M_{\Gamma,1} \) denotes memory at some time less than or equal to the time when \( M_{\Gamma,1} \) has been computed. This has the added difficulty that every \( M^P_i \in M_{\Gamma,1} \) 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 24.** 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. 19). If every communication statement begins a basic block, then any information transferred refers to the contents of memory at the start of the block \( M^P_{0i} \), and we can write:

\[ M^P_i = X_{\Gamma,1}(M_{\Gamma,0}, D^P_i) \]

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:

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

### 3.3. Parallel transition

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

**Definition 25.** 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 \( \sigma^P_j \in S_{G,J} \) and at least one serial state \( \sigma^P_k \in S_{H,K} \) such that there is a serial transition \( \sigma^P_j \rightarrow \sigma^P_k \) (at least one process takes a step).
iii. There are total states \( S_{G,J} \subseteq S_{\Gamma,L} \) and \( S_{H,K} \subseteq S_{\Gamma,M} \) such that \( \forall p \in \Gamma \), if \( s_j^p \in S_{G,J} \) and \( s_k^p \in S_{H,K} \) then \( 0 \leq k - j \leq 1 \). That is, states of individual processes that appear in both total states are separated by no more than one step. (Therefore there is a transition \( S_{\Gamma,L} \rightarrow S_{\Gamma,M} \) between total states that is compatible with or includes \( S_{G,J} \rightarrow S_{H,K} \).)

A parallel transition \( S_{\Gamma,J} \rightarrow S_{\Gamma,K} \) 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 \( G' \subset G \) and \( H' \subset H \), 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 \( H' \subset H \)).

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 \( \Gamma \) that may exist concurrently (the case in which each process is a subset with one member), however the actual set of subsets at any given time is still selected from \( 2^M - 1 \) possibilities.

3.3.1. Parallel state and communication. We have stated above that a particular parallel state \( S_{G,J} \) may or may not exist in reality, and that in fact a transition \( S_{G,J} \rightarrow S_{H,K} \) may actually include multiple partial transitions in which individual processes transit from process set \( G \) to process set \( H \). We need to consider the implications of this.

For simplicity, let us consider a transition \( S_{G,J} \rightarrow S_{G,K} \) in which the set of processes does not change. The indices \( J \) and \( K \) denote a set of step numbers for
processes in $G$ some or all of which may change during the transition, indicating a change in the basic blocks being executed by each process with a changed index. Suppose all processes in $G$ take a single step going from $J$ to $K$; unless they are tightly synchronized we must assume a large number of intermediate states in which some processes have advanced and some have not. It is even possible that some process that transit early into $S_{G,K}$ leave before other processes in $G$ enter the state.

The process set $G$ may be taken as defining the particular set of processes that execute the basic blocks denoted by the indices in $J$ and $K$, but not necessarily as a statement that all processes in $G$ are actually in the step numbers in $J$ or $K$ at the same time. In this sense, all state specifications are virtual, unless we insert synchronization code (such as a barrier) that forces processes to wait for each other in the state. For example, synchronous messages act like barriers, so we can state that states that include such messages must really exist.

### 3.4. 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 26.** 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:

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

$0$ denotes the multi-index in which every process 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 27.** A Parallel SPMD partial execution that does not diverge is given by a directed acyclic graph $E_{\Gamma, \Delta} = \langle V, A, S_{\Gamma, 0}, S_{\Gamma, End} \rangle$ 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 28.** 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 29.** 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 \( \text{End} \), then we have \((2^M - 1)^N\) possible executions with the same \( \Delta \) and \( \Gamma \).

**Definition 30. Ensemble:** We call any collection of executions of a particular program an ensemble of executions \( \Theta \). We will say that a data-process ensemble \( \Theta_{\Gamma, \Delta} = \{ E_{\Gamma, \Delta}^\varepsilon \} \) 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.

### 3.4.1. Properties of parallel execution

We are specifically interested in two properties of a data-process ensemble (same program, same data in the same sequence) \( \Theta_{\Gamma, \Delta} \):

#### 3.4.1.1. 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 31. 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 \( \sigma_p^j \rightarrow \sigma_{p+1}^j \).

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.

#### 3.4.1.2. 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 25.
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**

```plaintext
Case 1. 1. (0,1,*)
Case 2. 2. (1,*,0)
Case 3. 3. (*,0,1)
Case 4. 4. (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)
```
implies $(0,1,*)$, 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 $i \rightarrow ii \rightarrow iii$ we get:

**Case 1. Code for the Loop - first attempt**

```plaintext
Case 1. 1.a if(ME==0) MPI_ssend(x to 1 )
Case 2. 1.b if(ME==1) MPI_srecv(x from 0)
Case 3. 2.a if(ME==0) MPI_ssend(y to 2 )
Case 4. 2.b if(ME==2) MPI_srecv(y from 0)
Case 5. 3.a if(ME==1) MPI_ssend(z to 2 )
Case 6. 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 1 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 1. Code transformed to match 1
Case 2. 1.a if(ME==0) MPI_ssend(x to 1 )
Case 3. 3.a if(ME==1) MPI_ssend(z to 2 )
Case 4. 1.b if(ME==1) MPI_srecv(x from 0)
Case 5. 2.a if(ME==0) MPI_ssend(y to 2 )
Case 6. 2.b if(ME==2) MPI_srecv(y from 0)
Case 7. 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 - 1 depicts the pattern of sends and receives.

Case 1. Pattern of sends and receives in 1
Case 2. p0 → p2
Case 3. p0 → p1 → 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 1. Simultaneous statement pairs
Case 2. s1 [1.a@0 and 1.b@1]
Case 3. s2 [3.a@1 and 3.b@2]
Case 4. s3 [2.a@0 and 2.b@2]

Now, comparing 1 with 1 we can make the following statements about time sequences:

Case 1. Temporal dependences
Case 2. s1 < s2 because statement 1.a is before 2.a at process 0
Case 3. s2 < s3 because 2.b is before 3.b at process 2
Case 4. 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 1 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 → s2 → s3 → s1; that is, 'the Loop' from 1 in which the first state repeats.

Condition iii from the definition 25 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.
3.5. 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 21 would require that the particular sequence of parallel states be repeatable.

We will call this strict determinism.

Definition 32. **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 $\Theta_{\Gamma,\Delta}$ 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 21, 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 33. **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 $\Theta_{\Gamma,\Delta}$ has only one member.

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.
CHAPTER 4

Synchronization and determinism

Definitions

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 note that the following is not a standard definition, and leads to current and ongoing research. We may define synchronization as follows:

Definition 34. 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.

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

Definition 35. 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, 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:
4.1. SYNCHRONOUS TRANSITION

**Theorem 36.** Let \( p \neq q \) be processes, \( \ell_p^i, \ell_q^j \) be the values of a clock \( c \) at each of the states \( s_p^i, s_q^j \) of those processes. The relations \( \ell_p^i = \ell_q^j, \ell_p^i > \ell_q^j \) and \( \ell_p^i < \ell_q^j \) include all possibilities. In the first case, we say that the states \( s_p^i, s_q^j \) 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. □

**Corollary 37.** A synchronization \( \mathcal{S} \) 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: \( \leq, \geq, \neq \). □

1. = equal; exemplified by a barrier or a synchronous message.
2. < 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.

**4.1. Synchronous transition**

We now introduce a deterministic parallel transition, which we will use in exploring deterministic execution.

**Definition 38.** 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 \( S_{T,J} \Rightarrow S_{T,K} \).
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: $D_r = S_{r,0} \Rightarrow^* S_{r,End}$ 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.

### 4.2. Communications

As we have seen above (section 3.2.0.3), an execution in which all communication takes the form of messages is simpler (i.e. has fewer states) than one in which we allow one sided communication (put/get logic); this follows because the transformation that each basic block performs on memory takes a simpler form. Since determinism is one of the issues we have in parallel execution - that is, we want to insure that we will compute the same answer with the same data inputs, (or, more loosely, that we will compute one of a set of correct answers), we favor message passing communication.
Point to point messages

5.1. Messages and synchronization

We will assume for purposes of our discussion that we are able to measure the
times at which events occur using some kind of clock at each process, and that
these clocks are approximately synchronized - that is, that their time values are
approximately the same if examined concurrently.

A message from a sender process $p$ to a receiver process $q$ is a synchronization
between the two processes if it establishes a temporal order between events at each
process. It may be argued that this order is always determined by the direction of
data transfer - that is, that the clock at the sender is earlier than the clock at the
receiver. While this is obviously true for a physical message, the actual relation
between processes, and therefore the type of synchronization, is determined by the
blocking properties of send and receive statements.

Definition 39. A blocking communication statement is one that must com-
plete the specified action before the program can continue to the execution of the
next statement. Specifically, a blocking send completes when a matching receive
statement receives the transmitted message, and a receive statement completes
when the memory into which it places the message contains all the transmitted
data.

A non-blocking communication statement is a declaration that the process that
executes the statement is ready to send or receive a message, but it returns imme-
diately. Non-blocking communication may take place some arbitrary time after
the execution of the communication statement, and may require additional code to
verify that the message has actually been sent or received.

Buffered sends are usually considered non-blocking, in that execution can con-
tinue immediately after the message has been copied into a buffer. It is safe to
immediately re-use the memory location named in the send statement, because the
message has been copied to other memory. This is by no means the only way to
achieve a non-blocking send, however. For example, the mpi_isend non-blocking
send instruction does not use an additional buffer and returns immediately. It is
then necessary to use an mpi_wait or mpi_test instruction to verify completion of
the send.

A non-blocking receive instruction usually is accomplished by a pair of instruc-
tions, such as mpi irecv to declare that the receive can take place and specify the
memory location that holds the received data, and again mpi test or mpi wait to
verify that the data has been received and the memory location may be accessed.

Using Jordan’s [JORDAN03] classification, we have four possible combinations of blocking and non-blocking send and receive actions, given in table 1.
Table 1. Semantics of Messages

<table>
<thead>
<tr>
<th>send</th>
<th>receive</th>
<th>semantics</th>
<th>comment</th>
<th>implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>block</td>
<td>send=recv</td>
<td>data dep. constraints</td>
<td>synchronous</td>
</tr>
<tr>
<td>ii</td>
<td>non-b</td>
<td>block</td>
<td>s ≤ r</td>
<td>as above</td>
</tr>
<tr>
<td>iii</td>
<td>block</td>
<td>non-b</td>
<td>s ≥ r</td>
<td>as above, MPI asynch. receive</td>
</tr>
<tr>
<td>iv</td>
<td>non-b</td>
<td>non-b</td>
<td>any above</td>
<td>only data constraints, SOS runtime</td>
</tr>
</tbody>
</table>

Let process p be the sender and process q be the receiver.

Case i in table 1 implies a state $S_{G,J}$ (definition 22) in which the process group is $G = \{p, q\}$ and the step multi-index is $\{i, j\}$, at which the communications occurs. Since, for case i both send and receive are blocking, neither process is allowed to advance beyond $S_{G,J}$ until communication has occurred. We can take the process clock values at $\{p, q\}$ to be the multi-index values $\{\text{clock}_p, \text{clock}_q\} = \{i, j\}$. Normalized to $S_{G,J}$ the clock values at which communication takes place are $\{\Delta_p, \Delta_q\} = \{0, 0\}$; this is a barrier synchronization, $\Delta_p = \Delta_q$.

In case ii, the sender is not blocked, but the receiver is. Therefore p is allowed to advance while q is blocked, and we h communication occurs when $\Delta_p \geq \Delta_q$. Similarly, in case iii of blocking send but non-blocking receive we have $\Delta_p \leq \Delta_q$.

In case iv, the communication itself does not place added constraints on the clocks, although considerations of correctness would force the receiver process q to block and wait for communications to complete before reading the transmitted data (this also applies to case iii). On the sender side, if buffer space is limited, then the sender must be blocked if a buffer that has not been communicated needs to be re-used. Therefore all cases except i may require additional code to execute after the send and receive statements to block processes as needed.

Note that in we must always have $\{\Delta_p, \Delta_q\} \geq \{0, 0\}$ since communication cannot occur before the code that specifies it.

5.1. Semantics of send-receive pairs.

In every case, the semantics of send and receive requires that data appear to have been transferred when those statements execute. That is, if there is a send($x$) statement in the code which transmits the value of a memory location $x$, then writes to $x$ that appear in the text after the send can not affect the value that is sent, whereas writes to $x$ textually before send must affect the value transferred. Similarly, if there is a receive($y$) statement, any reads of location $y$ textually after the receive must get the transmitted value, but reads before the receive statement must get the original value.

The above considerations apply to every case in table 1, which leads to the belief that semantics of send and receive are independent of implementation. We show that this is not the case through schedule dependence graphs.

Definition 40. A schedule dependence between two statements $s_1$ and $s_2$ is a synchronizing relation between clocks that measure the execution of each statement, which relation is enforced by code or program logic.

We distinguish the following possibilities:

Theorem 41. A schedule relation between at two statements $s_1$ and $s_2$ must be one of the following:
5.1. MESSAGES AND SYNCHRONIZATION

i. Equals: \( \text{clock}_1 = \text{clock}_2 \). Holds if both statements are constrained to execute synchronously. Applies if both statements are blocking.

ii. Unequal: \( \text{clock}_1 \neq \text{clock}_2 \). Holds if either statement executes in a critical section.

iii. Greater: \( \text{clock}_1 < \text{clock}_2 \). Holds if \( s_2 \) is blocking but \( s_1 \) is not; \( s_2 \) may not execute until \( s_1 \) does.

iv. Independent: Holds if neither \( s_1 \) nor \( s_2 \) has to wait for the other; applies if neither \( s_1 \) nor \( s_2 \) is blocking.

**Proof.** A clock measures a scalar number (time), the only possible relations between a pair of scalars are \( = \), \( > \), \( \neq \), combinations of these or no relation.

If both statements are blocking, then each has to wait for the other and they execute at the same normalized clock time.

If one statement is executing in a critical section, then the other cannot execute and the normalized clock times must be unequal.

Properties of blocking and non-blocking follow definition 39.

Take a state \( S_{\{1,2\}\{i,j\}} \) such that \( s_1 \) is at process 1 in the block executed at step \( i \), \( s_2 \) is at process 2 in the block executed at step \( j \). Normalize clocks by \( \{-i, -j\} \) giving a (normalized) state \( S_{\{1,2\}\{0,0\}} \). If \( s_2 \) is blocking, it must wait for \( s_1 \) to execute. Since \( s_1 \) is non-blocking program execution can advance at process 1. Therefore event measured by \( \text{clock}_1 \) is earlier than event measured by \( \text{clock}_2 \).

If neither statement is blocking, the actual execution of both statements is determined by the runtime system and is not constrained by the actual communication statement code.

If \( s_1 \) is blocking and \( s_2 \), we invert indices in case iii so this does not add anything new; the given relations exhaust all possibilities \( \square \)

A schedule dependence graph represents scheduling (but not data) constraints between communication statements. There is a schedule dependence between a pair of statements at the same process if the first statement must complete execution before the second statement can start. This holds if the first statement is blocking, but not if the first statement is non-blocking (def 39). Schedule dependence between two communications at different processes is described by theorem 41 holds.

We denote a dependence by an arrow directed from a non-blocking statement that must execute in order to allow a blocking statement to complete its execution and advance (as in case iii of theorem 41). Note that, since the non-blocking statement allows program execution to continue, the more advanced clock is at the base of the dependence arrow.

If two clocks are related by a greater than or equal operator: \( \text{clock}_1 \geq \text{clock}_2 \) for \( s_1, s_2 \) we note from theorem 41 that, of the relations \( > \) and \( = \), the \( > \) relation imposes fewer restrictions on execution since it applies when only one statement is blocking; therefore we take the dependence arrow from \( s_1 \) to \( s_2 \) as above.

We construct a schedule dependence graph as follows:

**Algorithm 42. Construction of a communication schedule dependence graph**

**Input:** a list of processes; ordered lists of communication statements at each process, annotated with destination, source, and blocking or non-blocking properties.

**Output:** A directed graph \( S = \{V, A\} \) in which the arcs \( A \) denote schedule dependences and indicate restrictions on the execution order of communication statements. \( S \) denotes a partial order in which statements may be executed.
5.1. MESSAGES AND SYNCHRONIZATION

Procedure:
1. Initially, nodes in $V$ are individual communication statements, labeled with process numbers at which each statement is executed (If the same text is executed at multiple processes, it appears multiple times in $V$).

2. If $l_1, l_2 \in V$ are statements that are executed by the same process, $l_1$ appears before $l_2$, and $l_1$ is blocking, place an arc $l_1 \rightarrow l_2$ in $A$, indicating that $l_1$ must execute before $l_2$. Repeat until every consecutive pair of communication statements appearing in the list of statements for each process has been examined.

3. If a statement $c_1$ contains a send or receive which is matched by a send or receive in a statement $c_2$, and communications are synchronous (case i table 1) then merge nodes $c_1$ and $c_2$, indicating statements that must execute together. Repeat until no more nodes can be merged.

4. If a statement $c_1$ contains a send which is matched by a receive in a statement $c_2$, sends are non-blocking and receives are blocking (case ii table 1) then place an arc $c_1 \rightarrow c_2$ in $A$, since $\clock_1 \leq \clock_2$.

5. If a statement $c_1$ contains a send which is matched by a receive in a statement $c_2$, sends are blocking and receives are non-blocking (case iii table 1) then place an arc $c_2 \rightarrow c_1$ in $A$, since $\clock_2 \leq \clock_1$.

Repeat steps 4 and 5 until no new arcs are be added.

Given a schedule dependence graph as above, we can now state:

**Theorem 43.** A necessary condition for the execution of a set of communication statements is that their schedule graph be non-cyclic.

**Proof.** Assume false. A cycle in a dependence graph is one of the conditions of deadlock since every statement in the cycle requires that some other statement in the cycle be executed first and provides no starting point.

Note that, since the graph constructed by algorithm 42 takes only schedule into account, it is still possible that execution is prevented by a data dependence.

5.1.2. Examples. We now show through an example that the same syntax of send and receive statements leads to different schedule graphs.

Let processes be numbered $0 \ldots MAX$ for $N = MAX + 1$ processes. Let $ME$ represent the identity of the local process. Processes are part of an SPMD execution in which each process has its own local memory and variable names are replicated at each process. Consider now the following pseudo-code.

**Example 44.** At every process do:

```plaintext
send(x, to((ME + 1) modulo(N)));
receive(y, from((ME - 1) modulo(N)));
```

These statements take processes to be arranged in a ring; the modulo function replaces $MAX + 1 = N$ with $0$ and $0 - 1$ with $MAX$. The program logic takes a local value stored in $x$ at each process, and copies it to a memory location $y$ at the next process in the ring. To do this with `send` and `receive`, each process sends to the next process in the ring and receives from the previous process (taking next and previous to refer to numeric order).

We note that there are no conflicts due to data dependences, since the source value and the received value are held in different memory locations at each process, and neither location is read or written between the communication statements.
5.1. MESSAGES AND SYNCHRONIZATION

Figure 5.1.1. Blocking send/recv

Case 1: blocking send, blocking receive

Schedule diagrams 5.1.1, 5.1.2, 5.1.3 and 5.1.4 correspond to each of the four cases in table 1 as applied to three processes numbered 0 to 2 running the code of example 44. Although the data transfer appears does not change, we see that graphs 5.1.1 and 5.1.3 deadlock. We note also that graph 5.1.4 only shows communications schedule dependences; since all statements are non-blocking, there are none. However it may still be necessary to block processes in this case to satisfy data dependences; for example a process may need to be blocked after a non-blocking receive statement when the received data needs to be referenced.

Schedule diagrams 5.1.1, 5.1.2, 5.1.3 and 5.1.4 correspond to each of the four cases in table 1 as applied to three processes numbered 0 to 2 running the code of example 44. Although the data transfer appears does not change, we see that graphs 5.1.1 and 5.1.3 deadlock. We note also that graph 5.1.4 only shows communications schedule dependences; since all statements are non-blocking, there are none. However it may still be necessary to block processes in this case to satisfy data dependences; for example a process may need to be blocked after a non-blocking receive statement when the received data needs to be referenced.

We see in the example that the code fragment 44 has different execution dependent on the blocking constraints. We conclude that the semantics of the code in 44, and by extension the semantics of send and receive statements is different depending on the blocking behavior. It is our contention therefore that buffering, blocking and in general the underlying runtime system that supports messages is not simply a question of efficiency, but rather is one of the determining factors of message semantics.

5.1.3. Conflict between schedule and data dependences. A cycle in the schedule graph can often be resolved by a re-ordering of the communication statements. For example, the conflict in graph 5.1.1 can be resolved by switching the order of the send and receive statements in one process, yielding graph 5.1.5.
Figure 5.1.2. Non-blocking sends

Case 2: Non-blocking send, blocking receive

Figure 5.1.3. Non-blocking receives

Case 3: Blocking send, non-blocking receive

The conflict in graph 5.1.3 can be resolved by a somewhat counter-intuitive reversal of send and receive statements at all processes (graph 5.1.6). Note that in both cases, deadlock-free correct code under one set of blocking assumptions must be modified so that it is correct under a different set of assumptions.

So far we have limited ourselves to considering just schedule dependences, and figures 5.1.5 and 5.1.6 resolve these for code fragment 44. Suppose we modify...
Case 4: Non-blocking send, non-blocking receive

Figure 5.1.5. Deadlock resolved by schedule change

Case 1: Blocking send, blocking receive
Send–receive order inverted at P2

this code by adding a reference to one of the memory locations involved in the communications:

Example 45. At every process do:
send(x, to((ME + 1)modulo(N)));
x = some.function(y);
receive(y, from((ME - 1)modulo(N)));
Figure 5.1.6. Deadlock freedom with non-blocking receive

Now the solution given in 5.1.5 which involves swapping the send and receive actions at one process will not work; since we are updating x between the communication statements, we would send the wrong value of x at one process. The solution in 5.1.6 would replicate the same error at all processes. What we have in code fragment 45 is a straightforward pair of data anti-dependences (read before write) which, however, only work as stated in case ii from table 1; and in case iv only if the runtime system is somehow aware of the anti-dependence and blocks processes as needed to preserve them.

5.2. A standard send-receive semantics

A problem with the variety of semantic interpretations permitted by different blocking modes is that whether code is correct or not becomes uncertain. For example, is the code 44 correct? The MPI standard [MPI 95] considers this code to be correct but unsafe. Others would consider the code always incorrect because it may require unlimited buffer space. We feel that a standard semantics would be desirable; we would like to be able to talk about code as being correct or incorrect, not have to consider things like correct but unsafe, or incorrect but efficient.

Planguages [PLANG] adopt synchronous semantics, allowing buffering or blocking to become an implementation efficiency issue. This however forces us to write code that implements the dependence graph 5.1.5, even where resources exist that would allow us to express greater concurrency as in graph 5.1.1.

Synchronous semantics is safe, can run with minimal resources and helps to assure freedom from deadlock. Hoare [HOARE 85] shows that synchronous communication is sufficiently expressive for any computation. However, we still would prefer to be able to take advantage of resources that allow greater concurrency if they exist, so we hesitate to standardize on synchronous messages semantics.
5.3. NON-BLOCKING MESSAGING.

Although the authors have found non-blocking receives useful in writing real systems, we also find that thinking in terms of writing receives first and then sends is not particularly natural or intuitive. Again, we would not choose this as our semantic standard.

Non-blocking sends with blocking receives have the advantage of allowing the expression of more concurrency than synchronous send-receive pairs, and also coincide with our intuition about causality of messages. That is, they allow the sender process to perform a send action and continue, but require the receiver to wait for the message. However, the implementation of this mode requires either extra code (as in MPI asynchronous sends) or buffers. The buffered implementation appears simpler to use, but again leaves open the issue of safety. That is, without unlimited buffer space, running out of buffer or buffer overflow can occur in a program, and this would not allow us to guarantee correctness.

All of the blocking alternatives share the problem that the ability to execute correctly depends on schedule considerations imposed on top of data dependence considerations. We see in section 5.1.3 that code that appears to satisfy all data dependences can nevertheless be incorrect because of schedule problems.

5.3. Non-blocking messaging.

This brings us to the fourth alternative of non-blocking send and non-blocking receive. This has the desirable property that the only theoretical restrictions on the communications are that data dependences be preserved. The semantics of data transfer using this option are clear - all reads from, and writes to, variables involved in communication must be consistent with the send or receive having happened where the send or receive code appears in the program, similar to the semantics of put and get except that synchronization rather than being explicitly inserted would need to be deduced from the data dependences.

The difficulty is the implementation of this idealized form of non-blocking messaging. In practice most message systems require complex code to determine when communications have completed that in effect forces processes to block at the first point where a data conflict might arise (see examples of MPI asynchronous code in [MPI 95]).

We proposed an overlapping communications protocol in [GOMEZ 98], and we further develop and describe its implementation in [GOMEZ04]. The protocol requires the addition two new instructions: LH(memory_location) and RH(memory_location), used to indicate variable use to the system. The LH call is entered immediately before the next appearance of a variable on the left-hand side of an expression (for write access). The RH call is inserted before the next use of a variable on the right-hand side of an expression for read access. These are calls to the runtime system designed for automatic insertion by a compiler, but it is straightforward to enter them by hand as needed.

A communication instruction does the following: First it computes the communication pattern required to carry out the specified instruction, as a series of point to point sends. Then it assigns a Finite State Machine (FSM) to control the communication. Particular sends and receives required at the each process are inserted in a list maintained by the runtime system, local to each process. If the particular communication is a send, the runtime sends a status message to the intended receiver indicating that the particular send is ready to execute. If the communication
is a receive, the runtime system waits for a status message from the sender before proceeding.

The runtime system periodically examines all pending communications and carries out whichever sends or receives can proceed to completion, in any order - for example, sends for which matching receives are posted will execute before sends that have no matching receives, even if they were posted later. Similarly, receives will execute as their matching sends become available, not necessarily in the order in which they are posted.

The LH and RH instructions provide the system with information about variable use. If there is no pending communication with respect to a particular variable, the instruction is ignored. Otherwise an LH or RH code is given to the FSM that regulates the pending communication. Depending on the current state of the machine, this may cause the process to block.

The runtime system, however, remains active and continuously checking for status update from other processes. In this way, as matching sends or receives become available, all pending communications that do not violate data constraints can complete. Therefore the system resolves deadlocks that may arise from schedule conflicts that do not involve circular data dependence [GOMEZ04].

We term our system SOS, which stands for MIPS (formerly Ipstreams), Overlapping and Shortcutting. Shortcutting and overlapping were described in [GOMEZ 98]. MIPS (Merging Implicit Process Sets) are a construct that supports groups of processes formed implicitly by program logic, and are described in a paper currently in preparation. We are here concerned only with the semantic implications of overlapping, non-blocking communications.

The overlapping protocol, first described in [GOMEZ 98], refers to overlapping of sections of code between communication statement and variable use at different processes. Conceptually, it treats a communication statement as a declaration that communication can take place. If the pending communications is a send, read access to the variable being transmitted is allowed, but the process is blocked before any write access. If the pending communication is a receive, then either read or write access to the variable to be received causes the process to be blocked. Overlapping in this context refers to the possibility that the interval between the communication statement and variable use at a sender process may coincide at least in part with the equivalent interval at a receiver process. If this is the case, communication can take place without needing to block either process. The complexity of the FSM described in [GOMEZ 98] is mostly required to support collective communications in which a single variable may be involved in multiple sends and receives as part of a single instruction.

The system was designed originally to support irregular computation by performing this overlapping. We have since realized its extension to deadlock prevention [GOMEZ04] and its semantic implications.
CHAPTER 6

Collective communication and complexity

6.1. Point-point messages are sufficient

CSP (see C.A.R. Hoare - Communicating Sequential Processes) communicates entirely through “rendezvous” between processes - that is, fully synchronous point to point communications. CSP is a fully general concurrent language, it is in fact used in software engineering to specify concurrent systems. The fact that we can program anything in CSP tells us we don’t need any communications mechanism beyond message passing.

It is relatively simple to prove this claim. Consider a total state $S_{\Gamma,J} = \{\sigma_p^p, M_p^p\}$ of a concurrent execution. The set $\{M_p^p\}$ is the collection of memory contents at all processes in $\Gamma$. Now assume that $\{\sigma_p^p\}$, the basic blocks being executed, contains message code that simply copies $M_p^p$ from each process to some specific process, for example $p = 0$. Then, process 0 can perform any computable operation on $\{M_p^p\}$, since it has all the information, and transmit results back to all other processes via messages. Therefore anything computable by the full set of processes $\Gamma$ can be done using only message passing.

The above is a proof outline, but it can be formalized if needed. Also, note that we are not saying anything about efficiency - there may be methods that are more efficient than message passing, particularly if there is hardware support for them.

Even though point to point messages are sufficient, how we arrange our communications impacts how our programs run.

When we analyze sequential programs, it is important to estimate the amount of work required by an algorithm (time complexity, big O), and the space required (space complexity). There is frequently a trade-off between time and space complexity - i.e. we can save time by using more space. In fact, concurrent programming uses this tradeoff - we speed up our computations by using space and resources of many computers rather than just one. Similar considerations apply to communications, in particular when there is data to be transferred to and from multiple processes (as in the proof outline in section 1).

We consider two specific modes of collective communication (both used in the proof):

1. Broadcast: one process sends information to all other processes.
2. Reduction: one process needs to receive information from all other processes - this is frequently combined with broadcast in what MPI calls an allreduce; information is gathered at one process, computation is performed to produce a result at that one process, the result is then broadcast.
6.2. Broadcast

Designate some process \( p \) as root process, send data from \( p \) to all other processes. MPI_Bcast has \( p \), variable that holds data, communicator (e.g. MPI_COMM_WORLD) as parameters. All processes in the communicator execute the same MPI_Bcast statement, the process identified as root sends the data, all others receive it. How is this done? (By the way, in Planguages broadcast is also a single statement - it looks like \( x = y \oplus p \), where \( p \) is the root process - this means copy the value of \( y \) at \( p \) and put it into \( x \) everywhere).

Hardware support is an issue - if we have a data bus (for example an ethernet segment where many nodes are wired together without passing through a switch or router), or a wireless network, the root sends and all others receive. This does not scale well to large numbers of processes - we use switched networks. Therefore we need to send multiple messages.

Let root process be numbered 0, we have \( N \) processes numbered 0 to \( N - 1 \). Space complexity is the number of messages required. Time complexity is the number of steps required, where there may be multiple messages sent concurrently in one step. We assume that each process can send or receive one message per step - changing this assumption to any fixed number can speed things up, but only changes the constant not the complexity (as in big O complexity estimates).

Simple solution: root process sends in turn to every other process. We need \( N - 1 \) messages (space complexity); since each message is sent from 0 and it only sends one message per step, there are \( N - 1 \) steps (time complexity). Space and time communication complexity are both \( O(N) \).

Ring: ring networks (as in baton passing systems) are simple to build and analyze, we can set up a logical ring if processes send to \( (id + 1) \mod (N) \) and receive from \( (id - 1) \mod (N) \). Under this scheme, process 0 sends only one message, to 1; all others pass the message on until it reaches process \( N - 1 \). Space and time complexity are still \( O(N) \) - note that this is also the complexity of baton passing.

Tree: arrange processes in a tree (for example, a binary tree - each process has a right child \( 2 \times id + 1 \) and a left child \( 2 \times id + 2 \). We still need \( N - 1 \) total messages (number of edges in the tree) but now we get concurrent messages - it takes 2 steps for 0 to send first to 1 and then to 2, but 1 and 2 can then send concurrently to 3, 4, 5 and 6 (and in fact 1 can start immediately while 0 is still sending to 2). Therefore each level of the tree can send to the next level in 2 steps (because each parent node performs 2 sends, but all nodes in a level can send concurrently). Therefore the number of steps (time complexity) is \( 2 \times (height) \); the height of the tree is \( \log(N) \) (base 2 log for binary tree), so communication time complexity is \( O(\log(N)) \).

We can do more efficient trees; for example in a binary tree the root node does nothing while lower layers are active - we may want to give nodes higher in the tree more work to increase concurrency. Calculating a tree from a hypercube (see notes on networks) does this - nodes on each level of the tree have one less child than nodes on the next highest level. However, this only changes the constant; complexity is still \( O(\log(N)) \) because that is tree height.

Note that we are not limited to a root node of 0- one easy way of calculating the tree for any root is to put node numbers into an array, placing the node we want as root in index 0 - then we calculate a tree with root 0, but use the numbers in the tree as indices to the actual process numbers.
Trees appear to be the best we can do for broadcast, or for that matter for any collective communication based on point-point messages.

6.3. Reduction

A reduction is just a broadcast in reverse - the entire argument in the broadcast section is repeated, just reversing the message direction. In a tree, for example, we start in the leaves and work up to the root.

A reduction generally involves computing some binary function - that is, there is some function $y = f(a, b)$ and we want to compute a total $y$ for all processes. For instance, $f$ could be the sum of $a, b$ or the maximum. $f$ should not depend on order of its parameters: $f(a, b) = f(b, a)$ for reduction to make sense, although we can compute reduction for any binary function (Consider, what, if anything, does the difference of all $x$ values at all processes mean?). (Technically we want $f$ to be Abelian - don’t worry about this terminology, but you may come across it in more mathematical papers).

Syntax of reductions can be confusing - even though we have a function of 2 parameters, MPI reduce or allreduce only takes one variable; in Planguages we would write $y = f\{a\}$ for a reduction using function $y = (a, b)$. The idea is, the second parameter is the same variable at some other process.

In practice, reductions execute code something like:

- if (I am not leaf node)
- loop (each child node) my-value = f(my-value, value-from-child)
- if (I am not root) send (my-value to parent)

That is, each node gets values from its children, combines them with its own value and passes it up the tree. The root gets the final values and computes the final total. In an allreduce, the above code may be followed by a broadcast (on some networks, such as hypercubes, a faster allreduce may be implemented through a dimensional exchange algorithm - this has better parallelism but the same complexity as a tree-based reduction followed by broadcast).

6.4. Semantics, and why we care about it

When we argued above that $\{\sigma^p_j\}$ (set of basic blocks being executed) contains point-point message code that copies all of memory to a single process we were playing fast and loose with our own theory, because we have previously said that we are allowing only a single communication statement per basic block, and we know that it takes multiple steps (therefore multiple blocks) to get all the data to a single process. We could argue that, with multiple steps, the memory could change at each step and that therefore the root process does not actually get a true copy of the memory everywhere else.

That is one reason why I said we were doing a proof outline; we really need to have multiple steps that only communicate, without computation (as in BSP). We can argue about whether multiple steps make a difference or not, the proof is obviously cleaner if we can communicate in a single step. (In a proof that allows multiple steps, we also need to say that whatever computation occurs during each step can increase the speed, but not actually calculate anything not otherwise computable - this is what happens in our pseudocode for reduction, above).

Now, though, we have explained how to implement broadcast and reduction operations with point-point messages, and we have MPI and Planguage instructions
that implement both in a single statement - so we can actually write the whole proof algorithm from section 1 in a single block (using an allreduce statement), or at most 2 blocks (a reduction followed by computation at the root in the first block, a broadcast in the second). We show that the computation during reduction only increases speed, but if we insist on only computing at the root, we could write a function that passes an array indexed on process number, and just has each process put its local data into the array so the root gets an array with data from each process indexed by process number and can then compute using all data).

So the question becomes - is it legitimate to treat collective communication statements like allreduce as single communication statements, even though we know that they really represent a complicated distributed pattern of messages?

We have argued previously that the control flow graph gives us a structure that is consistent at all language levels because the compiler preserves the meaning. Now we are forced to acknowledge that this is not always the case - when a reduction - which is a single communication in a single block - gets translated to low level message code, it expands to multiple statements and multiple blocks surrounded by control logic (things like - “if i am an interior node then receive from child nodes and send to parent”). This may happen at compile time or in run time code, but in either case what actually executes is the complex set of statements not the high level simple code.

Semantics comes to our rescue (always assuming we trust the compiler and runtime libraries). The concept here is, it does not really matter what the actual executing code looks like, as long as it is a faithful translation that preserves the meaning (and function) of the high level code. We are justified in performing our analysis on the high level code as long as we can show that it is possible to translate into low level code without changing the meaning (we claim to have done this meaning-preserving translation for broadcast and reduction).

It is important to really analyze and if possible prove the semantics. If the actual code does not mean what is claimed for it, or if its meaning can change at execution (as is the case for the MPI standard send and receive code, see discussion of message semantics), then the program will behave in unexpected and possibly incorrect ways. In particular, we need to consider the theoretical implications of the code we write so that we can predict and understand how it will behave.

6.5. Implementation of collectives: allreduce in MPICH and PC
CHAPTER 7

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 at statement $s_2$, and $s_2$ appears after $s_1$ in program text.

Each of these dependences constrains $s_1$ 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 46.**

<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 ) ( \rightarrow 0 )</td>
<td>recv ( X ) ( 0 \rightarrow 1 )</td>
<td>recv ( X ) ( 1 \rightarrow 2 )</td>
</tr>
<tr>
<td>( s_2 )</td>
<td>( X = func(X) )</td>
<td>( X = func(X) )</td>
<td>( X = func(X) )</td>
</tr>
<tr>
<td>( s_3 )</td>
<td>send ( X ) ( 0 \rightarrow 1 )</td>
<td>send ( X ) ( 1 \rightarrow 2 )</td>
<td>send ( X ) ( 2 \rightarrow 0 )</td>
</tr>
</tbody>
</table>

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

\[
\begin{align*}
\text{if the send is non-blocking; otherwise we have: } & \quad s^0_1 \rightarrow s^0_2 \rightarrow s^0_3 \rightarrow s^1_1 \rightarrow s^1_2 \rightarrow s^1_3 \rightarrow s^0_1 \\
\text{where we use a double arrow to denote a send and receive each of which depends on the execution of the other statement.}
\end{align*}
\]

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 \( s^0_1 \) 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 47.**

<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 ) ( \rightarrow 0 )</td>
<td>recv ( Y ) ( 0 \rightarrow 1 )</td>
<td>recv ( Y ) ( 1 \rightarrow 2 )</td>
</tr>
<tr>
<td>( s_2 )</td>
<td>( X = func(X) )</td>
<td>( C = func(X) )</td>
<td>( C = func(X) )</td>
</tr>
<tr>
<td>( s_3 )</td>
<td>send ( A ) ( 0 \rightarrow 1 )</td>
<td>send ( B ) ( 1 \rightarrow 2 )</td>
<td>send ( C ) ( 2 \rightarrow 0 )</td>
</tr>
</tbody>
</table>

There is now no data dependence \( s_1 \rightarrow s_2 \) at any process, but there are still schedule dependences, which are described the same way as before: \( s^1_1 \rightarrow s^2_1 \rightarrow s^3_1 \rightarrow s^1_2 \rightarrow s^2_2 \rightarrow s^3_2 \rightarrow s^0_1 \)

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 \( s_2 \) or \( s_3 \) at the same process. We have:

**Example 48.**

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>( s_1 )</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>( s_2 )</td>
<td>( X = func(X) )</td>
<td>( C = func(X) )</td>
<td>( C = func(X) )</td>
</tr>
<tr>
<td>( s_3 )</td>
<td>send ( A ) ( 0 \rightarrow 1 )</td>
<td>send ( B ) ( 1 \rightarrow 2 )</td>
<td>send ( C ) ( 2 \rightarrow 0 )</td>
</tr>
<tr>
<td>( s_4 )</td>
<td>recv ( Y ) ( 0 \rightarrow 0 )</td>
<td>recv ( Y ) ( 0 \rightarrow 1 )</td>
<td>recv ( Y ) ( 1 \rightarrow 2 )</td>
</tr>
</tbody>
</table>

This looks like it should work; with non-blocking sends we do not have a circular dependence. We have data send-receive dependences:

\[
\text{if the send is non-blocking; otherwise we have: } \quad s^0_0 \rightarrow s^0_1 , s^1_1 \rightarrow s^2_1 \text{ and } s^2_2 \rightarrow s^0_3 . \quad \text{We also have schedule dependences } s^3_1 \rightarrow s^4_3 \text{ at each process. It appears that } s_3 \text{ can execute at each process without depending on anything that happens at any other process. However, if sends are blocking we have: } s^0_0 \leftrightarrow s^1_1 \text{ and } s^2_2 \leftrightarrow s^0_3 . \quad \text{That is, we have a forward data dependence, since } s^0_0 \text{, for example, sends data to } s^1_1 \text{, and a schedule dependence from } s^1_1 \text{ to } s^3_1 \text{ since a blocking send can’t complete before the receive executes. This gives us:}
\]
s₀ → s₀ ↔ s₁ → s₁ ↔ s₂ → s₂ ↔ s₀; which is a cycle as before.

MPI non-blocking sends depend either on system buffers or programmer defined buffers; the code in example 48 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 s₃ executes at the same clock time at every process. Since statement s₄ executes (everywhere) after s₃, the data that was sent has to be someplace (in the network? in a buffer?) between the execution of these two statements.

7.0.1. Rules for dependences: Dependences 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 = something; CIN >> X, PUT(X)) followed by a read from X (something = function(X); COUT << X, GET(X)). WRITE → READ
2. A send of X that matches a receive of Y. SEND → 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 s₁ → s₂ if s₂ follows s₁ 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 s₂ and s₁. In the case of a loop, the dependence may be different iterations of the loop and it may be impossible to switch the order.
2. There is a one-way schedule dependence s₁ᵖ → s₂ᵖ if statements s₁ at process p must execute in order for a statement s₂ at process q to execute. For example, a buffered send, where s₂ 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 s₁ᵖ ↔ s₂$q$ 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.

Notes to be added on:

- virtual states vs real states
- states with repeated sets of blocks - how to guarantee
- 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
CHAPTER 8

Networks:

We have so far paid no particular attention to how different nodes are connected. In a system with a small number of physical processors, many arrangements may be used.

Simplest is perhaps to connect multiple CPUs on a bus, possibly using the same architecture to create a large shared memory. Experience in building parallel computers has shown that in practice this arrangement runs into serious congestion problems somewhere between 30 and 40 processors. More recent experience (2008-2009) indicates that we may be able to go beyond this, we have seen up to 512 stream processors (essentially SIMD hardware) in a single graphics card. These however are specialized processors that require very little data transfer when used for their design purpose.

Small distributed clusters are also straightforward, with a number of separate multiprocessor nodes connected by a single router. This router is capable of making a direct connection between any permutation of pairs of computers in the cluster (e.g., if you have 8 computers the router can directly connect any combination of 4 pairs). The router supports full bandwidth connections with relatively low latency, although it adds delays when it is necessary to change which computer connects to which.

As systems scale up to hundreds, thousands, or even more nodes, these simple arrangements are no longer possible. We need to arrange the nodes into a network (note that in networks built up of multiple routers, the network is reconfigurable).

8.1. Definitions

A network is a directed graph consisting of set of nodes (processes) which are linked by directed edges (channels); it can be described as a set \( \{V, E\} \) where \( V \) is the set of vertices or nodes, usually numbered from 0, and \( E \) is a set of directed edges \( V = \{(i,j)\} \) such that \((i,j)\) is an edge from node \( i \) to node \( j \); it is drawn as an arrow. If a link between two nodes is bidirectional, it is indicated by a pair of edges \((i,j)\) and \((j,i)\), but it may be graphically represented as either a two-headed arrow or as a line.

We usually assume in the ideal case that each node corresponds to a separate processor, and each edge to a communication link between one processor and another. It is, however, possible to emulate one network on a different network. In such cases it is possible that several nodes (processes) are present on a single processor, and that several edges must connect over a single physical link. It is also possible that some processors contain no processes, some physical links are unused, and that what appears as a single edge in the graph is physically implemented as several links passing through intermediate processors.
8.3. SPECIFIC NETWORKS:

Obviously there can be large inefficiencies in emulating a network on physical hardware that does not match. An omega network is a network on which any other network can be emulated with a loss of efficiency no worse than \( \log(n) \), where \( n \) is the number of nodes in the network being emulated. That is, an algorithm that does work \( W \) to communicate on any given network requires no more than \( W\log(n) \) on an omega network.

Theoretical performance of a network is affected by the assumptions we make about it. In general, we assume messages of unit length (this is actually a reasonable assumption, given that long messages are usually split up into packets of constant length). We may assume that an edge can hold a message; this is not really a good match to reality, because an edge matches up to a wire. While a message will be on a wire for some length of time, the wire cannot hold it indefinitely. If the message is not picked up when it reaches a node, it will be lost.

A better assumption is that we have at least one buffer that can hold a message at each node, and that a message can pass along an edge from one node to another when there is an empty buffer at the destination node. Number of buffers and possible allocation of separate buffers for input and output affects the properties of the network, in particular as regards deadlock [JOHNSON 93], [JOHNSON 96].

Other properties that affect network performance: If an edge is bidirectional, can messages in both directions transit the edge at the same time, or can the edge only accommodate traffic in one direction at a time? Can a node send or receive messages on different edges at the same time, or does it need a separate step for each message on each edge (or for each copy of the same message on each edge)? The usual assumptions are: only one message in one direction on an edge at a time, separate step needed for each message or copy on each edge. Real hardware may in fact allow bidirectional links or use of multiple links at the same time, note that this will increase congestion and bandwidth required.

8.2. Network properties:

Some useful properties to describe networks:

- Diameter: the maximum least distance between any pair of nodes. (The number of edges on the shortest path between the two nodes that are farthest from each other). This is indicative of the number of steps required to get a message from anywhere in the network to anywhere else.
- Degree: the largest number of edges connected to any node. (If edges are directed we may speak of outdegree or indegree). This is related to the cost and difficulty of manufacturing a node
- Number of edges: indicative of the cost of the network.
- Size of cut set: smallest number of nodes which, if removed, divide the network into two disconnected graphs. This is indicative of possible congestion.

8.3. Specific networks:

\( N \) is the number of nodes in the network, \( k \) is the dimensionality. The following are characteristics of some common networks:
The following figures depict some common network types:

<table>
<thead>
<tr>
<th>Network</th>
<th>Diameter</th>
<th>Degree</th>
<th>Wires</th>
<th>Cut</th>
<th>k</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ring</td>
<td>$\frac{N-1}{2}$</td>
<td>2</td>
<td>$N$</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>2D grid</td>
<td>$N^{-2}$</td>
<td>4</td>
<td>$2(N-N^{-2})$</td>
<td>$N^{-2}$</td>
<td>2</td>
</tr>
<tr>
<td>3D grid</td>
<td>$N^{-3}$</td>
<td>6</td>
<td>$3(N-N^{-3})$</td>
<td>$N^{-3/2}$</td>
<td>3</td>
</tr>
<tr>
<td>2D torus</td>
<td>$N^{-2}/2$</td>
<td>4</td>
<td>$2N$</td>
<td>$2N^{-2}$</td>
<td>2</td>
</tr>
<tr>
<td>3D torus</td>
<td>$N^{-3/2}$</td>
<td>6</td>
<td>$3N$</td>
<td>$2N^{-3/2}$</td>
<td>3</td>
</tr>
<tr>
<td>Hypercube</td>
<td>$k = \log_2 N$</td>
<td>k</td>
<td>$k2^{k-1}$</td>
<td>$2^{k-1}$</td>
<td>k</td>
</tr>
<tr>
<td>Binary tree</td>
<td>$2(\log_2 N)$</td>
<td>3</td>
<td>$N-1$</td>
<td>1</td>
<td>-</td>
</tr>
</tbody>
</table>
8.4. Distributed systems - typical networks

Generally we do not have regular network structures such as those described above. Instead we have computers as end nodes with single bidirectional links to a switch, in a star topology with the switch at the center. In a typical local area network we have a single switch. Multiple switches may themselves be connected either directly to each other or in a hierarchic, tree-like arrangement through other switches.

Nevertheless, the concepts and terminology of the above networks can be applied. For example, a star network with a single switch at the hub can be modeled as a fully connected network with a diameter of 2 (counting a connection through
the switch as 2 links), in which each link from an end node is bidirectional, and the single center node can support \(\lfloor \frac{N}{2} \rfloor\) simultaneous links for \(N\) end nodes.
## Index

### A
- Absence of deadlock, 33
- allreduce, 53
- Amdahl's Law, 12
- Anti-dependence, 55
- Automatic parallelism, 7

### B
- back edge, 22
- basic block, 17, 18, 28
- baton passing, 52
- binary function, 53
- blocking communication, 41
- Broadcast, 51
- BSP, 12
- Buffered sends, 41
- bus, 59

### C
- CFG, 17, 20
- Circuit, 11
- clock, 38
- code block, 17
- communication schedule dependence graph, 43
- communication statement, 18
- compound natural loop, 24
- control flow graph, 20
- CSP, 12

### D
- data anti-dependences, 48
- data dependence, 44
- data-process ensemble, 34
- data-related dependences, 55
- Deadlock, 34
- Degree, 60
- Determinism, 37
- deterministic, 29, 32
- Diameter, 60
- Direct dependence, 55
- Direct dependences, 57
- divergence, 33
- DOM, 21

### E
- efficiency, 8
- Explicit parallelism, 8

### F
- Flynn's taxonomy, 11
- forward edge, 22
- function call, 20

### G
- general parallel transition, 32

### L
- leader, 18
- Leaders, 31
- LH and RH instructions, 50
- lock-step synchronization, 37
- Logic statements, 18
- logical ring, 52
- Loops, 20

### M
- message passing, 31
- MIMD, 11
- MIPS, 50
- MISD, 11
- multi-index, 30

### N
- natural loop, 22
- network, 59
- non-blocking communication, 41
- Non-blocking messaging, 49
- non-blocking receive, 41
- non-blocking send, 41
- non-uniform predicate, 30
- NUMA, 10
- Number of edges, 60

### O
- omega network, 60
- Other data dependences, 57
- overlapping communications protocol, 49

### P
- Parallel execution, 33
<table>
<thead>
<tr>
<th>Term</th>
<th>Page(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parallel SPMD state</td>
<td>29</td>
</tr>
<tr>
<td>parallel state</td>
<td>32</td>
</tr>
<tr>
<td>parallel transition</td>
<td>31</td>
</tr>
<tr>
<td>partial execution</td>
<td>33</td>
</tr>
<tr>
<td>partial states</td>
<td>29, 32</td>
</tr>
<tr>
<td>partial transitions</td>
<td>34</td>
</tr>
<tr>
<td>Planguages</td>
<td>48</td>
</tr>
<tr>
<td>PRAM</td>
<td>11</td>
</tr>
<tr>
<td>Procedural parallelism</td>
<td>17</td>
</tr>
<tr>
<td>Process Determinism</td>
<td>37</td>
</tr>
<tr>
<td>Process model</td>
<td>16</td>
</tr>
<tr>
<td>PUT/GET</td>
<td>10</td>
</tr>
<tr>
<td>R</td>
<td></td>
</tr>
<tr>
<td>random access</td>
<td>10</td>
</tr>
<tr>
<td>reducible CFG</td>
<td>21</td>
</tr>
<tr>
<td>reducible graph</td>
<td>22</td>
</tr>
<tr>
<td>Reduction</td>
<td>51</td>
</tr>
<tr>
<td>reduction</td>
<td>53</td>
</tr>
<tr>
<td>rendezvous</td>
<td>51</td>
</tr>
<tr>
<td>ring networks</td>
<td>52</td>
</tr>
<tr>
<td>router</td>
<td>59</td>
</tr>
<tr>
<td>S</td>
<td></td>
</tr>
<tr>
<td>scalable</td>
<td>13</td>
</tr>
<tr>
<td>schedule dependence</td>
<td>42</td>
</tr>
<tr>
<td>schedule dependence graph</td>
<td>43</td>
</tr>
<tr>
<td>Schedule dependences</td>
<td>57</td>
</tr>
<tr>
<td>schedule relation</td>
<td>42</td>
</tr>
<tr>
<td>Semantics</td>
<td>53</td>
</tr>
<tr>
<td>Semantics of Messages</td>
<td>42</td>
</tr>
<tr>
<td>semantics of send and receive</td>
<td>42</td>
</tr>
<tr>
<td>SEND/RECEIVE</td>
<td>10</td>
</tr>
<tr>
<td>Serial execution</td>
<td>28</td>
</tr>
<tr>
<td>Serial state</td>
<td>28</td>
</tr>
<tr>
<td>Serial transition</td>
<td>28</td>
</tr>
<tr>
<td>SIMD</td>
<td>11</td>
</tr>
<tr>
<td>SISD</td>
<td>11</td>
</tr>
<tr>
<td>Size of cut set</td>
<td>60</td>
</tr>
<tr>
<td>SOS</td>
<td>50</td>
</tr>
<tr>
<td>space complexity</td>
<td>51</td>
</tr>
<tr>
<td>Speedup</td>
<td>13</td>
</tr>
<tr>
<td>SPMD</td>
<td>11, 16</td>
</tr>
<tr>
<td>SPMD parallel execution</td>
<td>33</td>
</tr>
<tr>
<td>state specifications</td>
<td>33</td>
</tr>
<tr>
<td>strict determinism</td>
<td>37</td>
</tr>
<tr>
<td>subroutine</td>
<td>20</td>
</tr>
<tr>
<td>superscalar</td>
<td>9</td>
</tr>
<tr>
<td>synchronization</td>
<td>38</td>
</tr>
<tr>
<td>synchronization code</td>
<td>33</td>
</tr>
<tr>
<td>synchronous messages</td>
<td>33</td>
</tr>
<tr>
<td>synchronous semantics</td>
<td>48</td>
</tr>
<tr>
<td>Syntax of reductions</td>
<td>53</td>
</tr>
<tr>
<td>T</td>
<td></td>
</tr>
<tr>
<td>The Loop</td>
<td>35</td>
</tr>
<tr>
<td>time complexity</td>
<td>51</td>
</tr>
<tr>
<td>total execution</td>
<td>33</td>
</tr>
<tr>
<td>total transition</td>
<td>34</td>
</tr>
<tr>
<td>tree</td>
<td>52</td>
</tr>
<tr>
<td>two-sided communications</td>
<td>31</td>
</tr>
<tr>
<td>U</td>
<td></td>
</tr>
<tr>
<td>uniform value</td>
<td>30</td>
</tr>
<tr>
<td>V</td>
<td></td>
</tr>
<tr>
<td>virtual</td>
<td>33</td>
</tr>
<tr>
<td>W</td>
<td></td>
</tr>
<tr>
<td>Write-write dependence</td>
<td>55</td>
</tr>
</tbody>
</table>
Bibliography


[ALLEN 70] F. E. Allen, Control Flow Analysis, SIGPLAN, pp. 1-19, 1970


[DEO 74] Narsingh Deo; Graph theory with Applications to Engineering and Computer Science, Prentice-Hall 1974


[JOHNSON 93] R. Johnson, K. Pingali, Dependence Based Program Analysis, SIGPLAN ’93 PLDI, June 1993


BIBLIOGRAPHY


[LYNCH96] Lynch, Distributed Algorithms, Morgan Kaufmann 1996


