

Languages and Formalisms for Parallel Programming

Vivek Haldar
Information and Computer Science
University of California, Irvine

June 7, 2001

Abstract

We survey the ways in which parallelism can be *expressed* and *described* — namely, by means of *programming languages* and *formalisms*. We describe the design alternatives that various parallel programming languages have explored, such as various units of parallelism and communication mechanisms. We also survey a number of formalisms and calculi which have been proposed in order to give a formal semantics for these languages, and to understand parallelism in general.

Contents

| | | |
|----------|--|-----------|
| 1 | Introduction | 2 |
| 2 | Taxonomy of Parallel Languages | 2 |
| 2.1 | The Need for a Parallel Programming Language | 2 |
| 2.2 | The Unit of Parallelism | 3 |
| 2.2.1 | Processes | 3 |
| 2.2.2 | Objects | 4 |
| 2.2.3 | Statements | 5 |
| 2.2.4 | Functional Parallelism | 5 |
| 2.2.5 | Logical Parallelism | 6 |
| 2.2.6 | Collection-Oriented Languages | 7 |
| 2.3 | Communication and Synchronization | 7 |
| 2.3.1 | Message Passing | 8 |
| 2.3.2 | Synchronous or Asynchronous? | 8 |
| 2.3.3 | Rendezvous | 9 |
| 2.3.4 | Remote Procedure Call | 9 |
| 2.3.5 | Data Sharing | 9 |
| 2.3.6 | Distributed Data Structures: Tuple Spaces | 10 |
| 2.4 | Mapping Computations to Processors | 11 |
| 3 | Formalisms for Parallel Computing | 11 |
| 3.1 | Communicating Sequential Processes | 12 |
| 3.1.1 | Semantics | 13 |
| 3.2 | Calculus of Communicating Systems | 15 |
| 3.2.1 | Semantics | 16 |
| 3.2.2 | Equivalence Relations | 16 |
| 3.3 | The Chemical Abstract Machine | 17 |
| 3.3.1 | Formal Definitions | 18 |
| 3.3.2 | An Example: Describing CCS in the CHAM | 19 |
| 3.4 | Semantics for Tuple-Space Languages | 19 |

| | | |
|-------|--|-----------|
| 3.5 | Relations between formalisms | 20 |
| 3.5.1 | Ignoring Asynchronicity? | 20 |
| 4 | Conclusion | 21 |

1 Introduction

Language directly affects the way we think, and indeed, what we think. The constructs, abstractions and idioms provided by a programming language have a bearing on the ease with which can build programs, and the ease with which they can be understood and maintained.

Intimately related to languages is the issue of *semantics* — defining what a language *means*. We would like to do this in a formal and rigorous way, to make it clear what exactly we are talking about when using a language. The fact that parallelism is unintuitive (when compared to sequential computation) makes it all the more imperative that we formalize the semantics of parallel programming languages as much as possible.

This paper is organized as follows: in section 2 we provide a taxonomy of parallel programming languages, by classifying them according to the important design choices of unit of parallelism and method of communication. In section 3 we survey some of the more well-studied formalisms and calculi proposed for expressing and understanding parallelism and communication. This includes Hoare’s communicating sequential processes, Milner’s calculus of communicating systems, and Berry and Boudol’s chemical abstract machine.

2 Taxonomy of Parallel Languages

Numerous languages have been proposed and used for expressing parallelism and programming parallel machines. In this section we try to characterize the design space for parallel programming languages, and provide a classification of the various types of parallel programming. (See [BST89] for an extensive survey of parallel programming languages). The focus is on the various design choices and tradeoffs which must be made in the design of a parallel programming languages — we shall mention and discuss particular languages relevant to the feature under scrutiny, but avoid giving a long list of languages proposed, and what features they have.

2.1 The Need for a Parallel Programming Language

The first question that must be asked is : how are parallel programming languages different from sequential programming languages? Or, put another way, what characteristics or problems do parallel programming languages have to deal with, that sequential languages do not? There are basically three issues that distinguish parallel programming from sequential programming:

1. The use of multiple processors
2. The cooperation or communication among the processors
3. The possibility of partial failure

Ideally, programming support for implementing a parallel system should be able to deal with these three issues in some manner. This support may either be provided by the operating system, or by a language specially designed for parallel processing. Why chose one over the other?

When letting the operating system handle parallelism, applications are programmed in a sequential language extended with library routines that invoke operating system primitives. At first sight, this seems to offer the advantage of being able to easily extend an existing programming environment to handle parallelism, as well as achieving some degree of abstraction. But closer scrutiny reveals a number of problems.

One problem is that control structures and data types of sequential languages are usually inadequate for parallel programming. While simple operations like forking a new process are simple, more complex communication, such as receiving a message from one of several based on selection criteria which depend on the content of the message, are cumbersome to express and complicated to implement.

Another problem is sending complex data structures as part of a message. Since the operating system does not know how data structures are represented, it is difficult for it to serialize a data structure for transmission. A language would be able to do this automatically, since it would know about data structure layout.

Using a special language for parallel programming has many other advantages, such as readability, portability and static type checking. But most importantly, a language presents a programming model that is higher level and more abstract, and hopefully makes it easier and more logical to think about, write and understand parallel programs.

2.2 The Unit of Parallelism

A parallel application consists of essentially a collection of sequential computations communicating with each other. It is these sequential parts which can be done in parallel. They are essentially *units of parallelism*. There are many paradigms along which parallel programming languages can choose their unit of parallelism.

The concept of unit of parallelism is closely related to that of *grain* of parallelism. The grain is the amount of computation time between communications. Large-grain parallel programs spend most of their time doing computations and communicate infrequently; fine-grain parallel programs communicate more frequently. Fine-grain parallelism is best applied to closely coupled parallel systems, because in loosely coupled systems the communication overhead is prohibitive.

Typical units of parallelism are processes, objects, statements, expressions and logical clauses.

2.2.1 Processes

In most procedural languages for parallel programming, parallelism is based on the notion of a *process*. Different languages have different definitions of this notion, but in general a process is a logical processor that executes code sequentially and has its own state and data. A major advantage of this model is simply its familiarity. Since almost every prominent multitasking operating system is built around the notion of processes, programmers are familiar with it, and the constructs and programming idioms used to handle processes. And since multitasking operating system execute processes *conceptually* in parallel, using a process view of parallelism seems very natural.

Using processes as the unit of parallelism is largely *orthogonal* to the language used, because if the underlying operating system supports processes, it is likely that many languages available on that platform will have interfaces to create, use and manipulate processes.

The process view of parallelism is usually supported by libraries for invoking common operations easily. An example of such a library is Message Passing Interface (MPI) [GLS99], which is a widely used standard for writing message passing programs.

Processes are created either implicitly by their declaration, or explicitly by some `create` construct. With implicit creation, one usually first declares a process type and then creates processes by declaring variables of that type. Some languages with implicit creation fix the total number of processes at compile time. This makes the efficient mapping of processes onto physical processes easier, but imposes a restriction on the kinds of applications that can be implemented in that language, since it requires that the number of processes be known in advance.

Having an explicit construct for creating processes allows more flexibility. For example, the creation construct may allow parameters to be passed to the newly created process.

Processes also need to resolve the issue of *termination*. Usually, there are constructs for a process to terminate itself, and other processes (usually children of a process). Care must be taken not to try and communicate with a terminated process.

2.2.2 Objects

An *object* is a self contained unit that encapsulates both *data* and *behavior*, and that interacts with the outside world exclusively through some form of message passing. The data contained in an object is visible only within the object itself. The behavior of an object is defined by its *class*, which comprises a list of operations that can be invoked by sending a message to an object. *Inheritance* allows a class to be defined as an extension of another previously defined class. This approach is intended for structuring programs in a clean and understandable way, reflecting the structure of the problem to be solved.

Sequential object oriented languages are based on a model of *passive* objects. An object is activated when it receives a message from another object. While the receiver of the message is active, the sender is waiting for the result, so the sender is passive. After returning the result, the receiver becomes passive again and the sender continues. At any point in time, only one object in the system is active.

There are several ways to obtain parallelism in object oriented languages. One way is simply to extend the concept of using processes for parallelism. Smalltalk[GR83] includes the traditional notion of a process and provides two kinds of modules — processes and objects.

Another approach is to use the object itself as the unit of parallelism. Parallelism can be obtained by extending the sequential object model in any of the following ways:

1. Allow an object to be active without having received a message
2. Allow the receiving object to continue execution after it returns its result
3. send messages to several objects at once
4. allow the sender of a message to proceed in parallel with the receiver

The first two methods effectively assign a parallel process to each object, resulting in a model based on *active* objects. The last method can be implemented using asynchronous message passing, or by letting a single object consist of multiple threads of control.

This model has been criticized [CG89b, WKH92] as usually not being a good fit for parallel programming, since in most cases concurrency features were added

after the language was designed. Since objects are accessible only by way of the methods they define, which are invoked by other objects, concurrent object oriented programming essentially boils down to message passing.

Synchronization is a problem in the presence of inheritance. When a subclass inherits from a base class, programs must sometimes redefine the synchronization constraints of the inherited method. When using critical sections, a superclass cannot guarantee that all subclasses will follow the protocol for entering the critical section. These problems are compounded with multiple inheritance, because then it is possible for conflicts among various inherited methods. We know of no object oriented concurrent language which implements multiple inheritance.

Actors [Hew77, Agh86, Agh90] is not a language, but a construct which consists of a mail address and a behavior. Each mail address is associated with an incoming message queue. When a message arrives, the actor executes a *script*. The script accepts the message if it recognizes it; otherwise it is rejected. The script may send messages to other actors it knows, to itself (usually by creating a copy of itself), or to an actor created specifically to handle the message. The script also specifies a replacement behavior, which is a new actor with the same mailbox name that acts on the next message. The actor enhances concurrency when it specifies the replacement behavior before responding to the newly received message. Abcl/1[YSTH87] and Act1[Lie87] are languages based on the actor construct.

Examples of concurrent object oriented languages are Concurrent Smalltalk, and Emerald[ea88].

2.2.3 Statements

A straightforward way of expressing parallelism is to group together statements which can be executed in parallel. This method is easy to use and understand. Initiation and termination of parallel computations is well defined. The major drawback of this method is that it has little support for structuring large parallel programs.

Simply grouping together parallel statements can only create a fixed number of parallel executions. A more generic method is to use a parallel *loop* command, or a *parallel for* statement, in which all iterations of the loop are executed in parallel.

Occam [Ltd84] has support for both of these language constructs. Statements can be executed either sequentially:

```
SEQ
```

```
  S1;  
  S2;
```

or in parallel:

```
PAR
```

```
  S1;  
  S2;
```

An example of the parallel looping construct is:

```
PAR i=0 FOR n  
  a[i] = b[i] + c[i]
```

2.2.4 Functional Parallelism

Functional languages [Bac78] take the view that computation is simply definition and application of functions, in the mathematical sense[Bac78]. Functions do

not have side effects (they cannot modify global structures like in imperative languages) — they simply take an input and give an output ¹. This absence of side effects makes functional languages amenable to parallelization. It means that functions not having an input-output dependency between them can be executed in parallel. For example, in the expression:

$f(g(x, y), h(y, z))$

The evaluation of g and h can be done in parallel. Thus, we consider *functions* to be unit of parallelism.

This is very fine-grained implicit parallelism, and suited to architectures which support it. There are pitfalls. Indiscriminately executing all functions in parallel may not always be optimal. Communication overhead must be weighed against the gain achieved from parallelism. If a function does relatively little work, the overhead of communicating the result back to the caller will outweigh the saving in computation time. Ideally, the compiler should analyze the program and decide which functions to parallelize, and on which processor to perform each function call. Alternatively, the programmer may be given methods to specify this.

2.2.5 Logical Parallelism

Logic programming, represented by languages such as Prolog, offer many advantages for parallelism. In these, *logical clauses* are typically the unit of parallelism.

For example, the following code

```
A :- B, C, D
A :- E, F
```

means that A can be proved if all of B , C , and D can be proved, or if E and F can be proved. We can parallelize in two ways:

- Prove the two clauses, B, C, D and E, F in parallel, until one succeeds, or both fail. This is called *OR parallelism*.
- For each clause, prove each sub-theorem in parallel, until they all succeed, or any one fails. This is called *AND parallelism*.

Another interpretation of parallelism in logic languages can be in terms of processes. This is done by associating a process with every sub-theorem to be proved. Then, in our example, the process trying to prove A can be replaced by three processes trying to prove B , C and D respectively. Such processes are very light weight and similar in granularity to procedure calls in an imperative language.

Problems arise when logical variables are shared between more than one clause. For example, in

```
A :- B(X), C(X)
```

both B and C try to generate a value of X , and thus cannot be evaluated in parallel. One way to solve this problem is to explicitly specify which goals can write to shared variables, and which can only read them. This approach is taken in Concurrent Prolog [Sha87]. Another method is to solve dependent goals sequentially. Both compile-time analysis and run-time checks are used to determine if two clauses are independent.

¹though almost every functional language does make some concessions to the imperative world and provides constructs such as imperative assignment, along with a castigating footnote advising abstinence from their use

2.2.6 Collection-Oriented Languages

Several programming languages arising from diverse practical and theoretical considerations share a common high-level feature: their basic data type is an aggregate of other more primitive data types, and their primitive functions operate on these aggregates. Acting on large collections of data with a single operation is especially well suited to parallel programming and massively parallel computers. Such languages are called *collection-oriented* [SB91]. Common kinds of collections supported by these languages include sets, sequences, arrays, vectors and lists. Common collection operations include summing all the elements of a collection, permuting the order of the elements, and applying a function to all elements of the collection.

Many conventional languages, such as C and Fortran, supply an aggregate data structure (typically an array). However, the only primitive operations on these aggregates are accessors to single elements. This often forces the user to write explicit loops to operate on elements in an aggregate fashion. Precisely because collection-oriented languages eschew explicit loops, they are in most cases ideally suited for implementation on massively parallel machines: the parallelism inherent in the operations removes the need for sophisticated compiler analysis normally needed to uncover available parallelism.

When speaking in the context of massively parallel implementation collection-oriented languages are also called *data-parallel* languages [HJ86]. This is because the parallelism comes from applying a single operation over a potentially large set of data, in contrast to *control-parallel* languages, in which different operations are done in parallel.

2.3 Communication and Synchronization

Parallelism and communication are intimately related. Parallel units must communicate, coordinate and synchronize with each other to accomplish a task in concert. Thus parallel languages must give due attention to the issue of designing suitable communication methods.

There are broadly two types of coordination — *communication* and *synchronization*. When a process A requires some data from process B , there must be some means for them to communicate this data between them. Moreover, if either A is not ready to receive this data, or B is not ready to send it, then one or the other must know about it, so that it can wait, or take alternative action. So we see that communication and synchronization are closely related.

Communication introduces *non-determinism* into parallel programming. This is because it is possible for a process to wait for data from many other processes, rather than from one specific process. Since we cannot predict which process will have the data ready first, this is non-deterministic. Thus we need mechanisms and notations to express and control this non-determinism.

Parallel languages can be further broadly divided into two classes according to their communication mechanisms. *Logically distributed* systems are made up of multiple software processes that communicate by explicit message passing. This is because they have logically separate address spaces. In contrast, *logically non-distributed* systems have a global shared address space, and parallel processes communicate through shared data. Note that this is a logical categorization, and does not entail a particular *physical* model to support it. Logically distributed systems may be implemented on a physically shared address space, or logically non-distributed systems may be implemented on systems with physically separate address spaces.

Logically distributed systems use mechanisms such as message passing, rendezvous, remote procedure call (RPC), objects and atomic transactions. Logically non-distributed systems use mechanisms mostly based on implicit com-

munication, such as communication by functional results (used in functional languages), shared variables (parallel logic languages), and distributed data structures.

2.3.1 Message Passing

Many issues need to be resolved to implement a message passing mechanism — who sends it, what is sent, to whom it is sent, is receipt guaranteed, can more than one be accepted, and if so, how they are chosen or processed.

The simplest form of message passing is *point-to-point* messages sent from one process (the sender) to another (the receiver). This is usually reliable, the language runtime transparently managing acknowledgements.

In all message passing mechanisms, the sender initiates the interaction *explicitly*. But the receipt of the message may be either *implicit* or *explicit*. When it is explicit, the receiver executes some sort of *accept* statement specifying which messages to accept and what to do when a message arrives. With implicit receipt, the receiver automatically executes some code when a message is received, usually creating a new thread of control. Explicit receipt gives the receiver more control over the acceptance of message, since the receiver can be in many states, and accept different types of messages depending on which state it is in, or maybe even conditionally.

Naming is another issue in message passing. To whom does a sender want to send a message? From whom is the receiver willing to accept messages? Naming could be either *direct* or *indirect*. Direct naming is used to specify one specific process — it could be a static name, or an expression evaluated at run time. Such a mechanism is *symmetric* if both sender and receiver name each other. It is *asymmetric* if only the sender names the receiver. Note that implicit message passing is always asymmetric. Indirect naming involves an intermediate object, usually called a *mailbox*, to which messages are sent, and to which the receiver listens. The most simple mailbox is simply a global name. More advanced schemes could pass mailboxes as values which are part of messages.

2.3.2 Synchronous or Asynchronous?

By far the most important factor when discussing a message passing scheme is whether it is *synchronous* or *asynchronous*. In synchronous message passing, the sender is blocked until the receiver has accepted the message. This means that the sender and receiver not only exchange data, they also synchronize. With asynchronous message passing, the sender does not wait for the receiver to be ready to accept its message. The sender continues immediately after sending the message.

In the asynchronous model, as the sender does not wait for the receiver to be ready, there may be several *pending* messages at the receiver's end. Issues that need to be resolved are how these messages will be *buffered*, in what *order* they will be processed,² and how failure (buffer overflows) are dealt with.

In the synchronous model, there can be no more than one message pending from any sender process to a receiver. This relieves the problems due to buffering and flow control. The disadvantage is that this is less flexible than asynchronous message passing, because the sender always has to wait for the receiver to be ready to accept the message, even if the receiver does not have to return an answer.

²in real time systems this is a very critical issue because the order in which messages are processed must be able to minimize or do away with *priority inversion*, an effect in which a higher priority thread is made to wait because the message of a lower priority thread is ahead of it in the buffer of messages

2.3.3 Rendezvous

Point-to-point message passing has the drawback that it can only establish one way communication between two parties. But often it happens that interactions between processes is two way. Although this behavior can be simulated by two point-to-point messages, we would like to have a higher level mechanism to do this. The first of such mechanisms is *rendezvous*. It was popularized mostly by its use in the language Ada.

A rendezvous between two processes serves the purpose of both communication, and synchronization. It consists of: the entry declaration, the entry call, and an accept statement. The entry declaration and accept statement are on the server side, and the entry call is on the client side. Syntactically, an entry declaration is like a procedure declaration, having a name, and a list of formal parameters. An entry call is similar to a procedure call, naming the entry and the process containing it, and supplying the actual parameters. An accept statement for the entry contains a list of statements to be executed when the entry is called (it may be empty).

A rendezvous between two processes S and R takes place when S calls an entry of R , and R executes an accept statement for that entry. This is synchronous, so the first process which is ready to interact waits. Once synchronized, R executes the statements in the accept block. R has access to the input parameters supplied by S , and can assign values to the output parameters, which are passed back to S . Once the accept block is executed, the two processes can again begin to execute in parallel, and are no longer blocked.

2.3.4 Remote Procedure Call

Remote procedure call (RPC) is the second mechanism to affect two-way communication between two processes. This is very similar to a procedure call, but the calling and called processes are different. The input parameters are sent across to the callee, which executes the code of the procedure, and then sends back the output parameters to the caller. During this execution, the caller is blocked, and it executes again when it gets back the results from the callee. Like the rendezvous mechanism, this is also synchronous. The difference between the two is that in rendezvous the caller is unblocked as soon as the callee is done executing the accept block.

RPC could be either *transparent* or *nontransparent*. In transparent RPC the goal is to mimic a normal procedure call as closely as possible. This offers the significant advantage of being a familiar programming construct, and making it easy to port sequential programs. However, remotely executing a procedure cannot always provide the same semantics as a local procedure call. For example, pointers make no sense for an RPC. It is also very difficult to handle processor crashes. Firstly, it cannot be determined at what stage of the execution of the RPC a processor crashed, since the caller can get no communication back from a crashed processor. More seriously, if the RPC has side effects, and the crash occurred in the middle of its execution, recovering is not as simple as executing it again, because executing it twice may have unpredictable side effects.

2.3.5 Data Sharing

So far we have discussed approaches to communication which are used by logically distributed systems. Now we turn to logically non-distributed systems, in which all parallel units execute in a conceptually shared address space. Now communication need not take place via explicit passing of messages, but can be done by using shared data. The main idea is this: if two processes have access to the same variable, communication can take place by one process setting the variable, and the other process reading it.

Why try to use a shared data model on hardware which has physically separate address spaces? There are several advantages. While message passing generally transfers information between two specific processes, shared data is accessible by any process. Assignment to shared data conceptually has immediate effect, whereas there is a delay between a message being sent and received. The disadvantage of using shared data is that we require mechanisms to prevent many processes from trying to modify the same data — this is the well known *mutual exclusion* problem. This is an old problem, and has been studied extensively in the context of multitasking operating systems, in which many processes execute conceptually in parallel by time-slicing. Many mechanisms, such as semaphores, mutexes, and monitors[Hoa74] have been proposed to handle this.

2.3.6 Distributed Data Structures: Tuple Spaces

Distributed data structures are data structures that can be manipulated simultaneously by several processes[CGL86, CG89a]. This paradigm was first introduced in the language Linda [Car87, CG89b], developed at Yale. Linda uses a *tuple space* to express this.

A tuple space(TS) is logically shared memory, though it may be implemented on a physically distributed address space. The elements of the TS, called *tuples*, are ordered sequences of values, similar to records. For example,

```
[‘parallel’, 1, 0.4]
```

is a tuple with three fields: a string, an integer, and a real.

The following operations are defined in a tuple space:

- **out** adds a tuple to the TS
- **in** reads and removes a tuple from the TS
- **read** reads a tuple from the TS

Tuples do not have addresses, but are referenced by their contents. They are denoted by specifying either the *value* or *type* of each field. Values are actual parameters, types are formal parameters. A tuple matches if each field matches either the value or type given in that field. If more than one tuple matches, one is chosen arbitrarily. If there are none, the operation and its invoking process block until the tuple space has a match. For example,

```
read(‘parallel’, int x, float y)
```

x is assigned the value 1 and y is assigned the value 0.4, assuming the tuple above is in the TS.

There is no operation to modify a tuple in place. To modify a tuple, it must be removed (using in), modified, and then put back in the TS (using out). Since the primitive operations out, in, and read are *atomic*, this neatly solves the mutual exclusion problem. This makes it possible to build distributed data structures using tuples. For example, a distributed array could be created using tuples of the form

```
[‘name’, index, value]
```

Note that the three primitive tuple operations are *orthogonal* to an existing language. They can be added to an existing language to make it into a parallel programming language. Indeed, this was the approach followed by the Linda creators. They extended a number of well established languages(C, Fortran) to create parallel languages (C-Linda, Fortran-Linda).

2.4 Mapping Computations to Processors

So far we have seen what the various units of parallelism in a parallel programming language could be. This ranges from fine-grained parallelism (such as in functional languages) to broad-grained parallelism (when using processes). A related issue is how these parallel computations are distributed over the available physical processors, i.e., which parallel unit is executed on which processor at a given point in time. This assignment of computations to processors is called *mapping*.

The mapping of processes to processors is similar to *load balancing* in distributed operating systems. There is one basic conceptual difference though. An operating system tries to achieve *fairness* in scheduling processor time and allocating resources to various processes and users. It may try to reduce communication costs by having frequently communicating processes run on the same processor. The goal of mapping, however, is to minimize the execution time of a single parallel program. Fairness is not an issue because parallel units are part of the same program, and are cooperating, not competing. The reduction of communication overhead achieved through mapping processes to the same processor must be weighed against the resulting loss of parallelism.

An important choice in the design of a parallel language is whether mapping will be under user control. If not, mapping is done transparently by the compiler and language run-time system, possibly assisted by the operating system. This may seem to ease the task of the programmer. But the system generally does not have any knowledge about the problem being implemented, so problem specific mapping strategies cannot be applied. This is a severe restriction.

Programmable mapping usually consists of two steps. First, the parallel units are mapped onto the physical processors. Several parallel units may be mapped onto the same processor. Second, the units mapped to the same processor are scheduled by a local mapping, usually based on *priorities* assigned to the parallel units.

There are several approaches to mapping parallel units to physical processors, whether it is done by the compiler or programmer. The mapping could be:

- fixed at compile time
- fixed at run time
- not fixed at all

3 Formalisms for Parallel Computing

Given the vast variety of programming languages and models which we have surveyed so far, the question arises: is it possible to understand and express parallelism with a few simple primitives? Can we formulate a rigorous, mathematical basis for it? How can we reason about parallel programs, or prove that they satisfy certain properties?

A parallel (no pun intended) can be drawn here with similar developments in sequential programming. Back in the 1970s, Scott, Strachey and many others built on the work of Turing and Church to precisely define the semantics of sequential programming [Sto77]. Much of that work utilized Church's lambda calculus [Chu51] as a cornerstone. The lambda calculus is striking in its simplicity — it has but three rules for constructing valid lambda expressions, and another three for rewriting expressions. A precise definition of the calculus can be given in less than half a page. Yet it is powerful enough to capture all sequential computation.

The question we must ask is: is there a formalism for parallel computation, which can be used to reason about, understand and give a rigorous foundation for our understanding of parallelism? As we will see in the following sections, many have been proposed.

This brings us to another question: if there are many formalisms, how are they different? And how are they similar? Are they related? Can one be reduced to another? Is there hope of finding a single formalism which subsumes them all, or, alternatively, proving that some of them are equivalent to others?

Again we can draw a parallel with sequential computation. There were two major mathematical models of computation — the Turing machine, and Church’s lambda calculus [Chu51]. They were proved equivalent to each other. The Church-Turing *conjecture* states that any other model of computation will also be equivalent to them. This equivalence between two models of computation which were discovered independently is very satisfying because it indicates that there is a sound and consistent mathematical basis for computation.

However, most practitioners rarely, if ever, use formal methods when writing sequential programs. That is because the meaning of sequential programs, and what they do, is intuitively clear to us. The von Neumann model of fetching and modifying words in a central store is easy to understand and intuitively reason about.

This is not the case for parallel programming. It is extremely hard to form an intuitive understanding, let alone informally reason about, of a program in which many threads are running in parallel and coordinating with each other. The addition of *nondeterminism* makes it doubly hard to talk about parallel programs informally. Thus, aside from the motivation to have a mathematically sound basis for what we are talking about, there is a dire need of formalisms for parallel programming due to purely pragmatic reasons.

In the following sections, we give brief overviews of some of the more well-known and studied formalisms proposed to describe parallel computation. We also try to examine the relationships between these various formalisms.

3.1 Communicating Sequential Processes

Communicating sequential processes was first proposed by Hoare in [Hoa78], and elucidated in greater detail in [Hoa85]. It is a language built around two simple, and related, primitives: *parallel execution*, and *communication*. The fundamental concept is, as the name suggests, sequential processes which communicate with each other using well-defined communication primitives. The language was designed in a minimalist way in order to study the *semantics* of parallelism. Here we briefly outline the major language constructs of CSP.

Processes may have *process labels* to identify them. For example,

```
P :: x := x + 1;
```

denotes the process P with the command list following the label.

A *parallel command* (\parallel) specifies concurrent execution of its constituent processes. They all start simultaneously, and the command terminates if and when all of them successfully terminate. The relative speeds of the various processes is arbitrary. Indices can be used to create a number of processes. For example, $X(i : 1..n) :: P$ means:

$$X(1) :: P_1 \parallel X(2) :: P_2 \parallel \dots \parallel X(n) :: P_n$$

Input and output commands specify communication between two concurrently operating sequential processes. Communication occurs between two processes of a parallel command whenever:

1. an input command of one process specifies as its source the process name of the other process
2. the target variable of the input command matches the value denoted by the expression of the output command.

Then the input and output commands are said to *correspond*. Corresponding commands are executed simultaneously, their effect being to assign the value of the expression of the output command to the target variable of the input command. $?$ is the input operator, and $!$ the output operator. For example,

$$X?y$$

means “from process X , input a value and assign it to variable y ”, and

$$X!z$$

means “to process X , output the value z ”

Non-determinism is introduced and controlled by using a *guarded select* construct, similar to the one introduced by Dijkstra [Dij85]. A guarded command (of the form *guard* \rightarrow *commandlist*) is executed only if and when the execution of its guard does not fail. First its guard is executed and then its command list. The constituents of a guard are executed from left to right. If a boolean expression evaluates to false, the guard fails; an expression evaluating to true has no effect. For example, in

$$x > y \rightarrow a := 1$$

the assignment happens only if the guard evaluates to true.

An alternative command consists of many guarded commands, and specifies execution of exactly one of its guarded commands. If all guards fail, the alternative command fails. In case more than one guard is successful, one is chosen arbitrarily for execution. Guarded alternative lists are separated by \square . An example of a guarded alternative command is:

$$[x \geq y \rightarrow m := x + 1 \square x = y \rightarrow m := x - 1]$$

Note that when $x = y$, two guards will be true.

The language also allows *input guards* — and input command at the end of a guard is executed only if and when a corresponding output command is executed.

Communication between processes must be done strictly by means of the input and output primitives. A process must be *disjoint* from other processes — it is not allowed to use (read or write) any variable which occurs as the target of an assignment in any other process.

Many languages have been deeply influenced by CSP (especially its notion of synchronous communication), the most notable being Occam[Ltd84].

3.1.1 Semantics

When CSP was first proposed in [Hoa78], it was simply presented as a new language, with lots of examples to explain how its parallel and communication primitives could be used, but lacking a formal semantics. This shortcoming was rectified a few years later in [BHR84], in which a denotational semantics for CSP is given. Soundarajan gives an axiomatic semantics for CSP in [Sou84]. Here we shall focus on the denotational semantics, mostly following the presentation given in [BHR84]. We need some definitions first.

The smallest unit in the behavior of a process is an *event*. They are instantaneous, and have no duration. The duration of elapsed time between any two

events is not considered, only the relative order in which they occur. Hereafter, A denotes the set of all events. The behavior of a process upto some moment in time is recorded as the sequence of all events in which it has participated — this is known as its *trace*.

Let s be a trace, and P and Q processes. A *transition* is:

$$P \rightarrow^s Q$$

which means that s is a possible trace of the behavior of P upto some moment in time, and that the subsequent behavior of P may be the same as that of Q . Transitions satisfy the following laws (\in is the empty transition):

$$\begin{aligned} P \rightarrow^s Q, Q \rightarrow^t R &\Rightarrow P \rightarrow^{st} R \\ P \rightarrow^{st} R &\Rightarrow \exists Q : P \rightarrow^s Q, Q \rightarrow^t R \\ P \rightarrow^\infty Q, Q \rightarrow^\infty P &\Leftrightarrow P = Q \end{aligned}$$

Processes can make internal progress, i.e., they can change their state without engaging in any externally visible event: $P \rightarrow^\infty Q$. Since processes are non-deterministic, such internal progress can only reduce the future non-determinism of P .

The *initials* of a process P , $initials(P)$ are those in which it can engage on the very first step. Let X be the set of events possible for some environment of a process P . Then if $initials(P) \cap X$ is empty, nothing further can happen. If P makes internal progress to Q , and $initials(Q) \cap X$ is empty, we say that X is a *refusal* of P . The set of all such X is called the set of *refusals* of P . If s is a trace of P , and after engaging in s , P can refuse the finite set X of events, the pair (s, X) is called a *failure* of P , $failure(P)$.

Processes are identified by their *failure sets*. So:

$$failure(P) = failure(Q) \Rightarrow P = Q$$

Why this negative characterization? There are two types of properties we are interested in proving: *safety* (“nothing bad will happen”), and *liveness* (“something good will happen”). While positive characterizations are good for proving safety, they are cumbersome to use when proving liveness properties. This negative characterization, rather than characterizing on the basis of traces alone, helps us to reason about what *must* happen.

Given these definitions, we shall outline the way in which the semantics of CSP is given in [BHR84]. Here we explain the semantics of two kinds of parallel composition operators — parallel composition by intersection, and by interleaving.

Parallel composition by intersection of two processes P and Q , denoted $P||Q$, behaves like both P and Q progressing in parallel. An event can occur only when both P and Q can participate in it simultaneously. Thus, we have:

$$P \rightarrow^s P', Q \rightarrow^s Q' \Rightarrow (P||Q) \rightarrow^s (P'||Q')$$

The process defined is :

$$P||Q = \{(s, X \cup Y) | (s, X) \in P, (s, Y) \in Q\}$$

Parallel composition by interleaving of processes P and Q , written $P|||Q$, also behaves like P and Q running in parallel, but differs greatly from $P||Q$ in that each event requires participation from only one of the processes, rather than both. Thus, each trace of $P|||Q$ is an interleaving on a trace of P and a trace of Q :

$$P \rightarrow^s P', Q \rightarrow^t Q' \Rightarrow (P|||Q) \rightarrow^u (P'|||Q')$$

where u is an interleaving of s and t . The process defined is:

$$P ||| Q = \{(u, X) | \exists s, t : (s, X) \in P, (t, X) \in Q, u \in \text{interleave}(s, t)\}$$

The internal progress transition \rightarrow^ϵ reduces the non-determinism of a process. Thus, it can be thought of as a *partial order* on the space of processes corresponding to a measure on non-determinism. The maximal elements of this ordering are the deterministic processes. It can be shown that this partial order gives the structure of a *complete semilattice* on the space of processes.

If F is a function from processes to processes, it can be shown that both $||$ and $|||$ are distributive, continuous, associative, and commutative in the semilattice of processes.

3.2 Calculus of Communicating Systems

The calculus of communicating systems (CCS) was proposed by Milner [Mil89]. Though similar to Hoare's CSP in appearance, Milner's motivations were different, and much more mathematical. While Hoare wanted to propose something close to a programming language, Milner took a much more mathematical and abstract approach, similar to that of Lambda calculus. Milner's calculus is motivated by questions such as: what primitives are needed to express concurrency? How can we say one concurrent system is equivalent to another? What criteria are used to judge such equivalences? The focus on *equivalence* between various concurrent systems is important because it highlights what aspects of a system we consider important and would like to see invariant across multiple real implementations. Indeed, the notion of equivalence tries to define what a concurrent system is.

CCS agents are built up from the most simple element, actions, which are defined as an atomic step of execution. Some of these actions refer to internal computational steps, some to interactions with other agents.

Let L be the set of *names*, and L' the set of *co-names*: $L' = \{a' : a \in L\}$. Note that $a'' = a$. Let τ denote the *internal communication* action. Then the set of actions is $A = L \cup L' \cup \{\tau\}$.

A name and a co-name form a pair of input and output ports - a channel. Because CCS is a binary synchronous modelling language, there can only be one input and one output port with the same name that are involved in the same synchronisation (that is, there can be more than one input and output ports with the same name, but only one of each can be allowed to be involved in a single synchronization). Note that the internal communication actions, τ actions, are not visible to the outside world

Agents are composed from the sets of actions that have just defined through a set of operators whose definitions follow. Formally, if A and B are agents, and a an action, then the following are also agents (0 is the *null* agent):

- $a.A$ is an agent that can do action a , and then behave like A .
- $A + B$ is the *non-deterministic choice* between A and B — it can behave either like A or B .
- $A|B$ is the *composition* of agents A and B .
- A/X Restriction. This operator hides all symbols (including complements, where they exist) in X from the agent definition A , where X is a subset of the labels.
- $A_{[new/old]}$ replaces all occurrences of *old* in A with *new*.

3.2.1 Semantics

An operational semantics can be given for CCS, which defines how a system will behave and evolve. It has rules of the form

$$\frac{\text{premises}}{\text{conclusions}}$$

So long all the premises are satisfied (above the line) then it is valid to draw the given conclusions (below the line). Here we explain some of the rules.

Prefixing has the following rule:

$$\frac{}{a.A \rightarrow^a A}$$

which says that the agent $a.A$ will, upon performing action a , behave like agent A .

The rule for choice is:

$$\frac{A_j \rightarrow^a A'_j}{\sum_{i \in I} A_i \rightarrow^a A'_j}$$

which says that from a set of choices defined in some indexing set I , a choice is made between them by performing some action.

There are three rules for composition:

$$\frac{A \rightarrow^a A'}{A|B \rightarrow^a A'|B}$$

$$\frac{B \rightarrow^b B'}{A|B \rightarrow^b A|B'}$$

$$\frac{A \rightarrow^c A' \quad B \rightarrow^{c'} B'}{A|B \rightarrow^\tau A'|B'}$$

The first rule says that given a choice between A and B performing some action, if A has an action that it may perform, it may do so. The second rule says the same for B . This may happen irrespective of whether other agents have valid actions. The third rule says that if two agents are synchronizing or communicating via a mutual channel (in this case c), then they must do so atomically, with both agents performing an action synchronously in one indivisible step — τ . It is this rule that restricts CCS to binary synchronizations, since τ' is not defined.

3.2.2 Equivalence Relations

We would like to analyze when two systems are the “same”, i.e. identify properties which we would like to remain invariant across many real implementations. There may be many ways of defining when we consider two systems to be the same, and this is captured by defining various *equivalence relations*. Here we briefly and informally sketch the various equivalence relations defined in CCS. For a rigorous formal treatment, see [Mil89].

The first equivalence relation is *strong bisimulation*. The basic idea of a strong bisimulation relation is that two systems should be able to pass through equivalent states, such that one can simulate the other’s behavior at all times.

What this says about the relation is that there must be a mapping between states such that each state can be related to one state in the other system.

The problem with strong bisimulation as defined above is that it is too inflexible with respect to internal computations (τ actions), in so far as one τ in one system that is not in the other will make the systems inequivalent.

To solve this, another bisimulation relation is introduced - a *weak bisimulation* relation. The definition is similar, although subtly different. It essentially allows any number of τ operations to be removed or inserted before or after any action. There is a problem with weak bisimilar relations - they do not form a congruence relation. This is because initial τ actions could be matched by an empty transition, thereafter making two systems different. Strongly bisimilar relations do, however, but they are too strict.

To solve this, we introduce our third type of bisimulation relation, *observational congruence*, which prevents initial τ actions being matched to an empty transition

3.3 The Chemical Abstract Machine

The Chemical abstract machine (CHAM), proposed by Berry and Boudol [BB90], is an abstract machine which tries to capture parallelism by emulating the physical process of chemical reactions. It builds on earlier work on the Γ language [BCM88, BM86]. Abstract machines such as the Turing machine and Random Access Machine have been widely used to study sequential programming and algorithms. Abstract machines are a middle ground between formulating and analyzing algorithms on a particular real hardware platform, and using theoretical models very far removed from real machines. Abstract machines try to capture the essential characteristics of real machines, while doing away with arbitrary low level details of a particular real machine. The goal is to have a model that can be applied to many real machines without loss of generality.

Most concurrency models are based on architectural concepts, such as a network of processes communicating by means of ports or channels. This imposes a rigid geometrical view of concurrency. Since parallel programming with control threads is considered difficult, a high level parallel programming methodology should be liberated from control management. The goal was to have a model where the concurrent components are freely “moving” in the system and communicate when they come in contact.

Intuitively, the state of a system is like a *chemical solution* in which floating *molecules* can interact with each other according to *reaction rules*. An unspecified mechanism stirs the solution, allowing for possible contacts between molecules. Under this model, the process of solution undergoing change by various reactions is clearly truly parallel — any number of reactions can be performed in parallel, provided that they involve disjoint sets of molecules.

Solutions are represented by *multisets* (a set which allows duplication of elements) of molecules — this accounts for the associativity and commutativity of parallel composition. The reaction rules are multiset rewritings.

Some molecules cannot interact with others. Those which can are called *ions*. A solution can be *heated* to break complex molecules into smaller ones, upto ions. Conversely, a solution can be *cooled* to rebuild heavy molecules from components. Also, to deal with abstraction and hierarchical programming, a molecule may have a subsolution contained in a *membrane*, which could be *porous* to allow communication between the encapsulated solution and its environment.

All chemical abstract machines obey a simple set of structural laws. Each particular machine is given by adding a set of simple rules that specify how to produce new molecules from old ones. Note that these rules are different from

inference rules typically used in operational semantics — they have no premises and are purely local. The notion of a membrane is particularly powerful, for it allows us to build chemical abstract machines that have the power of traditional process calculi, or concurrent generalizations of lambda calculus.

As an example, consider the following: the solution is made up of all integers from 2 to n . The reaction rule is that any integer destroys its multiples. Then the solution will end up containing the prime numbers between 2 and n .

3.3.1 Formal Definitions

A chemical abstract machine or CHAM C is specified by defining *molecules* $m_1, m_2 \dots$, *solutions* $S_1, S_2 \dots$, and *transformation rules*. Molecules are terms of algebras. Solutions are finite multisets of molecules, written $\{m_1, m_2, \dots, m_k\}$. Any solution S can also be considered as a single molecule, and can therefore be a *subsolution* of another molecule. The corresponding $\{\}$ operator is called the *membrane operator*. Transformation rules have the form

$$m_1, m_2, \dots, m_k \rightarrow m'_1, m'_2, \dots, m'_l$$

where m_i and m'_j are molecules.

The multiset union of S and T is written $S \uplus T$. The context notation $C[\]$ denotes a molecule with a hole \square in which to place another molecule. The transformation rules determine a *transformation relation* $S \rightarrow S'$ between solutions. The laws are:

- **The Reaction Law:** if

$$m_1, m_2, \dots, m_k \rightarrow m'_1, m'_2, \dots, m'_l$$

is a rule, and $M_1, M_2, \dots, M_k, M'_1, M'_2, \dots, M'_l$ are instances of the m_i and m_j , then

$$\{M_1, M_2, \dots, M_k\} \rightarrow \{M'_1, M'_2, \dots, M'_l\}$$

- **The Chemical Law:** Reactions can be performed freely within any solution:

$$\frac{S \rightarrow S'}{S \uplus T \rightarrow S' \uplus T}$$

- **The Membrane Law:** A subsolution can evolve freely in any context:

$$\frac{S \rightarrow S'}{C[S] \rightarrow C[S']}$$

- **The Airlock Law:** An airlock is a molecule of the form $m \triangleleft S$ where m is a molecule, and S a solution. Airlocks can be built and suppressed:

$$\{m\} \uplus S \leftrightarrow \{m \triangleleft S\}$$

A CHAM is an intrinsically parallel machine. Several rules can be applied to a solution if the molecules are involved in no more than one rule. Subsolutions can also be transformed in parallel.

Rules can be of three types: *heating rules* \rightarrow , *cooling rules* \rightarrow , and *reaction rules* \rightarrow . This is just a matter of classification, and not formalized. Heating rules decompose single molecules into simpler ones, and cooling rules recombine a compound molecule for its components.

3.3.2 An Example: Describing CCS in the CHAM

In [BB90] the authors illustrate the descriptive power of the chemical abstract machine by showing how to handle a subset of Milner's CCS[Mil89]. Here we sketch some of the rules of that CHAM — for complete details see [BB90].

Let $N = \{a, b, \dots\}$ be a set of *names* of $L = \{a, a' \mid a \in N\}$ be the set of *labels* built on N . α, β range over labels, with $\alpha'' = \alpha$.

The CCS agents are molecules which react with each other in a solution. The two basic rules are:

- *parallel*

$$p|q \rightleftharpoons p, q$$

This is simply the heating-cooling rule pair for parallel composition in CCS, which says that two parallel agents in a CCS term can become independent molecules in a solution, and vice versa.

- *reaction*

$$a.p, a'.p \rightarrow p, q$$

Since a is the ion's communication capability, it is called its *valence*. When two complementary solution are in a solution, they can react with each other and release their bodies, making the valences disappear.

To execute an agent p , we start with the solution $S_0 = \{p\}$. Heating frees up ions, exhibiting potential communications, which can be done using the reaction rule. As an example, let us try to reduce the agent $a.b.0|a'.0|b'.0$:

$$\begin{aligned} & \{a.b.0|a'.0|b'.0\} \\ \rightarrow^* & \{a.b.0, a'.0, b'.0\}(\textit{parallel}) \\ \rightarrow & \{b.0, 0, b'.0\}(\textit{reaction}) \\ \rightarrow & \{0, 0, 0\}(\textit{reaction}) \end{aligned}$$

The multitude of null agents (0) can be easily cleaned up with a rule which says that null agents simply evaporate:

$$0 \rightarrow$$

3.4 Semantics for Tuple-Space Languages

Linda introduced the new concept of using a tuple space and distributed data structures in the mid and late 1980s. This is also sometimes called *generative communication*. It was a very successful language and implemented on a number of commercial parallel hardware platforms. However, serious attempts to rigorously define the semantics of Linda did not appear in the literature until the late 1990s [COW97, CGZ96, Cam96, CJY95].

Campbell et al [COW97] provide a survey of the various attempts to provide a semantics for Linda, and explain the various choices that can be made. Some of the issues that need to be rigorously resolved are:

- Do active tuples (those created by `eval()` to create a new process) actually exist as tuples, or just as processes? Can they be handled like normal, passive tuples?
- Is `eval()` the only means of process creation, or can the host language's constructs also be used for creating other processes?
- How are tuple spaces organized? Is there a single, global tuple space? Can there be more than one tuple space? How are they named and specified? Can they be scoped, or exist in a hierarchy?

- How is garbage collection done for tuple spaces? What if a process which was the only one to hold the names of certain tuple spaces dies?
- Is input-output done using tuple spaces, or through the host language's usual means?

The fact that so many important questions are left open highlights the need to lay down a rigorous and formal semantics for Linda and Linda-like languages.

In [CGZ96] a semantics for tuple-space manipulation is given by embedding it in CCS. The advantage of this is that formal techniques developed for CCS can be adapted and used for reasoning about generative communication. Note that generative communication is *asynchronous*, while CCS is fully synchronous. The major difficulty here is to represent the communication medium. This problem is solved by having a *fully distributed* communication manager. More precisely, each tuple or process is modeled as an active CCS agent. A function is used to map Linda operations to CCS actions.

In [CJY95], an operational semantics for Linda is given using various schemes: Plotkin's structural operational semantics, CCS, Petri Nets and the chemical abstract machine.

3.5 Relations between formalisms

Given that many formalisms such as CSP, CCS, CHAM etc, exist for describing and expressing parallelism, it is natural to ask if they are related. Do they all have a common core? Can some of these formalisms be reduced to others? Can equivalences be proved among them? Unfortunately, most comparisons between various formalisms are of an informal nature.

Hoare informally mentions some similarities between CCS and CSP in [Hoa85]. For example, the following similarity between CCS and CSP notations (respectively) is pointed out:

- $a.P$ corresponds to $a \rightarrow P$
- $(a.P) + (b.Q)$ corresponds to $(a \rightarrow P | b \rightarrow Q)$

3.5.1 Ignoring Asynchronicity?

It is interesting to note that two of the most well-developed formalisms, CCS and CSP, both make the choice of modelling only synchronous binary communication. In [Hoa85] Hoare clearly outlines his reasons for preferring synchronous message passing over the buffered asynchronous case:

- It closely matches physical realization with circuits and wires
- When buffering is needed, it can be simulated by a process, letting the programmer precisely control how exactly it is done
- Bounding the length of buffers opens opportunities for deadlocks — thus we need unbounded buffers. This complicates both implementations and mathematical modeling.

There seems to be no formalism which tries to begin with *asynchronous* communication as a primitive. Perhaps the chemical abstract machine could be used to model asynchronous communication neatly by modeling an asynchronous message as an ion floating around in a solution, ready to react with some agent which is willing to accept the message. However, this is still overly simplistic, and issues such as the order in which messages will be processed, preferential treatment of some messages, and buffering of messages seems to be hard to model with a chemical abstract machine.

4 Conclusion

We have surveyed some of the major points in the design space of parallel programming languages. This includes issues such as what should be chosen as the unit of parallelism, what are the various mechanisms to communicate and synchronize between parallel units, and how they are mapped to actual processors.

We then take a look at the closely related issue of how to formally and rigorously define, express and reason about parallelism, concurrency and communication. This is done using various calculi and formalisms such as Hoare's Communicating Sequential Processes, Milner's Calculus of Communicating Systems, and the Chemical Abstract Machine proposed by Berry and Boudol. Lastly, we briefly look at some recent attempts to give a semantics for tuple-space languages such as Linda.

References

- [Agh86] Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986.
- [Agh90] Gul Agha. Concurrent object-oriented programming. *Communications of the ACM*, 33(9):125–141, 1990.
- [Bac78] John Backus. Can Programming be Liberated from the von Neumann Style? A Functional Style and its Algebra of Programs. *Communications of the ACM*, 21(8):613–641, August 1978.
- [BB90] G. Berry and G. Boudol. The chemical abstract machine. In *Proceedings of the seventeenth annual ACM symposium on Principles of programming languages*, pages 81–94, 1990.
- [BCM88] Jean-Pierre Banatre, Anne Coutant, and Daniel Le Metayer. A parallel machine for multiset transformation and its programming style. *Future Generation Computer Systems*, 4:133–144, 1988.
- [BHR84] S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe. A theory of communicating sequential processes. *Journal of the Association for Computing Machinery*, 31(3):560–599, 1984.
- [BM86] Jean-Pierre Banatre and Daniel Le Metayer. A new computational model and its discipline of programming. Technical Report 566, INRIA, 1986.
- [BST89] Henri E. Bal, Jennifer G. Steiner, and Andrew S. Tanenbaum. Programming languages for distributed computing systems. *ACM Computing Surveys*, 21(3):261–322, 1989.
- [Cam96] Duncan Campbello. On a chemical abstract machine for linda. Technical Report YCS 273, 1996.
- [Car87] N. Carriero. *The Implementation of Tuple Space Machines*. PhD thesis, December 1987.
- [CG89a] N. Carriero and D. Gelernter. How to Write Parallel Programs: A Guide to the Perplexed. *ACM Computing Surveys*, 21(3):323–357, September 1989.
- [CG89b] N. Carriero and D. Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, April 1989.

- [CGL86] N. Carriero, D. Gelernter, and J. Leichter. Distributed Data Structures in Linda. In *13th ACM Symposium on Principles of Programming Languages*, January 1986.
- [CGZ96] P. Ciancarini, R. Gorrieri, and G. Zavattaro. Towards a Calculus for Generative Communication. In E. Najm and J. Stefani, editors, *Proc. IFIP Conf. on Formal Methods for Open Object-Based Distributed Systems*, pages 289–306, Paris, France, 1996. IFIP Wg 6.1.
- [Chu51] Alonzo Church. *The Calculi of Lambda-Conversion*. The Annals of Mathematical Studies 6. Princeton University Press, 1951.
- [CJY95] P. Ciancarini, K. Jensen, and D. Yankelewich. On the Operational Semantics of a Coordination Language. In P. Ciancarini, O. Nierstrasz, and A. Yonezawa, editors, *Object-Based Models and Languages for Concurrent Systems*, volume 924 of *Lecture Notes in Computer Science*, pages 77–106. Springer-Verlag, Berlin, 1995.
- [COW97] D. Campbell, H. Osborne, and A. Wood. Characterising the design space for linda semantics. Technical Report YCS 277, Department of Computer Science, University of York, 1997.
- [Dij85] E. W. Dijkstra. Guarded commands, nondeterminacy, and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, August 1985.
- [ea88] E. Jul et al. Fine grained mobility in the emerald system. *ACM Transactions on Computer Systems*, pages 109–133, February 1988.
- [GLS99] Willian Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI*. MIT Press, 1999.
- [GR83] A. Goldberg and D. Robson. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley, 1983.
- [Hew77] C. Hewitt. Viewing control structures as patterns of passing messages. *Journal of Artificial Intelligence*, 8(3):323–364, June 1977.
- [HJ86] W. Daniel Hillis and Guy L. Steele Jr. Data parallel algorithms. *Communications of the ACM*, 29(12), December 1986.
- [Hoa74] C. A. R. Hoare. Monitors: An Operating System Structuring Concept. *Communications of the ACM*, 17(10):549–557, October 1974.
- [Hoa78] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978.
- [Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.
- [Lie87] Henry Lieberman. Concurrent object oriented programming in act 1. In Akinori Yonezawa and Mario Tokoro, editors, *Object Oriented Concurrent Programming*, pages 9–36. MIT Press, 1987.
- [Ltd84] Inmos Ltd. *Occam Programming Manual*. Prentice Hall, 1984.
- [Mil89] Robin Milner. *Communication and Concurrency*. Prentice Hall, 1989.
- [SB91] Jay M. Sipelstein and Guy E. Blelloch. Collection-Oriented Languages. *Proceedings of the IEEE*, 79(4):504–523, April 1991.

- [Sha87] E. Shapiro. *Concurrent Prolog: Collected Papers*. MIT Press, 1987.
- [Sou84] N. Soundararahan. Axiomatic semantics of communicating sequential processes. *ACM Transactions on Programming Languages and Systems*, 6(6):647–662, 1984.
- [Sto77] Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press, 1977.
- [WKH92] Barbara B. Wyatt, Krishna Kavi, and Steve Hufnagel. Parallelism in object-oriented languages: a survey. *IEEE Software*, 9(6):56–66, November 1992.
- [YSTH87] Akinori Yonezawa, E. Shibayama, T. Takada, and Y. Honda. Modelling and programming in an object oriented concurrent language abcl/1. In Akinori Yonezawa and Mario Tokoro, editors, *Object Oriented Concurrent Programming*, pages 55–90. MIT Press, 1987.