Coordination of Cooperative Agents

F. Arbab

1. Introduction

The ever decreasing costs and sizes of processors, their ever increasing speeds, faster and wider-band-width communication links, and global networks have made the potential of applying the computational power of several (even hundreds and thousands) of processors to a single application, a reality. Conceptually, the significance of the availability of so many processors to work on an application goes beyond performance issues.

The mere idea of allocating more than one *worker* to the same task immediately opens up a new problem solving paradigm, and simultaneously, presents a new challenge. The paradigm is *concurrency*, and the challenge is *coordination.*

We consider the problem of coordination of very large numbers of concurrent active entities that must cooperate with each other in the context of a single application. We give a brief problem description and history in section 2. In section 3 we distinguish between *communication* and *cooperation*, and show the need for a coherent model and language to describe the cooperation protocols of active entities in massively concurrent systems. Next two models are compared, the currently most widely used *Targeted-Send/Receive*, or TSR model and a new coordination model, *Idealized Worker Idealized Manager* (IWIM) model. A specific coordination language [1,2], called MANIFOLD, that is based on the generic model proposed in section 3 is described in section 4. Some of the interesting features

of **MANIFOLD** are shown through examples in this section. The conclusion of the paper is in section 5.

2. CONCURRENT PROGRAMMING

Concurrency is about the expression of a computation as a set of concurrent activities. As such, it is a problem solving or a programming paradigm. Parallelism is about allocating more resources to carry out a given computation. As such, it is a method for realizing a solution, i.e., to carry out a computation.

Of course, the study and the application of concurrency in computer science has a long history. The study of deadlocks, the dining philosophers, and the definition of semaphores and monitors were all well established by the early seventies. However, it is illuminating to note that the original context for the interest in concurrency was somewhat different than today in two respects:

- In the early days of computing, hardware resources were prohibitively expensive and had to be shared among several programs that had nothing to do with each other, except for the fact that they were unlucky enough to have to compete with each other for a share of the same resources. This was the *concurrency of competition.*
- The falling costs of processor and communication hardware only recently dropped below the threshold where having very large numbers of 'active entities' in an application makes sense. Thus, it is no more unrealistic to think that a single application can be composed of hundreds of thousands of active entities. This is the *concurrency of cooperation.* Compared to classical uses of concurrency, this is a jump of several orders of magnitude in numbers, and in our view, represents (the need for) a qualitative change.

Theoretical work in this area, e.g., π-calculus or process algebra, is still too fundamental to be used directly in large concurrent applications. On the other hand, the tried and true models of cooperation, such as client-server, barrier synchronization, etc., that are used in practical applications of today are simply a set of ad-hoc, special-case templates; they do not constitute a coherent paradigm for the definition of cooperation protocols.

We believe there is a clear need for programming models that explicitly deal with concurrency of cooperation among very large numbers of active entities that comprise a single application. Such models cannot be built as extensions of the sequential programming paradigm. Because such applications can be distributed over a network, we believe such models cannot be based on synchronous models of concurrency.

3. Communication vs. cooperation models

It is important to distinguish between the conceptual model describing the cooperation of a number of concurrent processes in an application, and the underlying model of communication on top of which such cooperation is implemented [3].

The primary concern in the design of a concurrent application must be its model of cooperation: how the various active entities comprising the application are to cooperate with each other. Eventually, a communication model must be used to realize whatever model of cooperation application designers opt for, and the concerns for performance may indirectly affect their design. Nevertheless, it is important to realize that the conceptual gap between the system supported communication primitives and a concurrent application must often be filled with a non-trivial model of cooperation.

There is no paradigm wherein we can systematically talk about cooperation of active entities, and wherein we can compose cooperation scenarios. Consequently, programmers must directly deal with the lower-level communication primitives that comprise the realization of the cooperation model of a concurrent application. Because these primitives are generally scattered throughout the source code of the application and are typically intermixed with non-communication application code, the cooperation model of an application generally never manifests itself in a tangible form – i.e., it is not an identifiable piece of source code that can be designed, developed, debugged, maintained, and reused, in isolation from the rest of the application code.

3.1. The TSR model of communication

A common characteristic of most flavours of the message passing model of communication is the distinction between the roles they assign to the two active entities involved in a communication: the sender and the receiver. A sender s typically sends a message m to a receiver r. The send operation is generally targeted to a specific (set of) receiver(s). A receiver r, on the other hand, typically waits to receive a message m from any sender, as it normally has no prior knowledge of the origin of the message(s) it may receive. We use the term *Targeted-Send/Receive* , or TSR, to refer to the communication models that share this characteristic.

Consider the following simple example of a concurrent application where the two active entities (i.e., processes) p and q must cooperate with each other. The process p at some point produces two values which it must pass on to q. The source code for this concurrent application looks something like the following:

process p:	process q:
compute m1	receive m1
send m1 to q	let z be the sender of m1
compute m2	receive m2
send m2 to q	compute m using m1 and m2
do other things	send m to z
receive m	
do other computation using m	

The first significant point in the above listing is that the communication concerns are mixed and interspersed with computation. This decreases the understandability, the maintenability and re-usability of the cooperation model.

The second significant point to note is the need of specifying a target for the send and the asymmetry between send and receive operations. Targeted send strengthens the dependence of individual processes on their environment. This too diminishes the reusability and maintenability of processes. In this model, debugging and proving programs correct are also not trivial: a process is not a well-encapsulated concept into this model, it sometimes needs the existence of some other valid processes to be valid.

3.2. The IWIM model of communication

In the following, we consider an alternative generic model of communication that, unlike the TSR model, supports the separation of responsibilities and encourages a weak dependence of workers (processes) on their environment. We refer to this generic model as the *Idealized Worker Idealized Manager* (IWIM) model.

The basic concepts in the IWIM model are *processes*, *events*, *ports*, and *channels*. A process is a *black box* with well-defined ports of connection through which it exchanges *units* of information with the other processes in its environment. A port is a named opening in the bounding walls of a process through which units of information are exchanged using standard I/O type primitives analogous to read and write. Without loss of generality, we assume that each port is used for the exchange of information in only one direction: either into (input port) or out of (output port) a process. We use the notation $p.i$ to refer to the port i of the process instance p.

The interconnections between the ports of processes are made through channels. A channel connects a (port of a) producer (process) to a (port of a) consumer (process). We write $p.o \rightarrow q.i$ to denote a channel connecting the port o of the producer process p to the port i of the consumer process q.

Independent of the channels, there is an event mechanism for information

exchange in IWIM. Events are broadcast by their sources in their environment, yielding an *event occurrence*.

The IWIM model supports *anonymous communication*: in general, a process does not, and need not, know the identity of the processes with which it exchanges information. This concept reduces the dependence of a process on its environment and makes processes more reusable.

A process in IWIM can be regarded as a worker process or a manager (or coordinator) process. The responsibility of a worker process is to perform a (computational) task. A worker process is not responsible for the communication that is necessary for it to obtain the proper input it requires to perform its task, nor is it responsible for the communication that is necessary to deliver the results it produces to their proper recipients. In general, *no process in* IWIM *is responsible for its own communication with other processes*. It is always the responsibility of a manager process to arrange for and to coordinate the necessary communications among a set of worker processes.

There is always a bottom layer of worker processes, called *atomic workers*, in an application. In the IWIM model, an application is built as a (dynamic) hierarchy of (worker and manager) processes on top of this layer. Note that a manager process may itself be considered as a worker process by another manager process. Let us reconsider the example in subsection 3.1., and see how it can be done in the IWIM model. A new process c responsible to facilitate the communication has been created. The source code looks something like the following:

```
process p:                          process q:                          process c:

compute m1                          read m1 from input port i1          ...
write m1 to output port o1          read m2 from input port i2          create the channel p.o1 ⟶q.i1
compute m2                          compute m using m1 and m2           create the channel p.o2 ⟶q.i2
write m2 to output port o2          write m to output port o1           create the channel q.o1 ⟶p.i1
do other things                                                         ...
read m from input port i1
do other computation using m
```


In this example, the responsibility of the coordinator process c is, very simple: perhaps, it first creates the processes p and q, establishes the communication channels defined above, and then may wait for the proper condition (e.g., termination of p and/or q) to dismantle these channels and terminate itself.

Nevertheless, moving the communication concerns out of p and q and into c already shows some of the advantages of the IWIM model. The processes

p and q are now 'ideal' workers. They do not know and do not care where their input comes from, nor where their output goes to. They know nothing about the pattern of cooperation in this application; they can just as easily be incorporated in any other application, and will do their job provided that they receive 'the right' input at the right time. The cooperation model of this application is now explicit: it is embedded in the coordinator process c. If we wish to have the output of q delivered to another process, or to have yet another process deliver the input of p, neither p nor q, but only c is to be modified.

The process c is an 'ideal' manager. It knows nothing about the details of the tasks performed by p and q. Its only concern is to ensure that they are created at the right time, receive the right input from the right sources, and deliver their results to the right sinks. It also knows when additional new process instances are supposed to be created, how the network of communication channels among processes must change in reaction to significant event occurrences, etc. (none of which is actually a concern in this simple example).

Removing the communication concerns out of worker processes enhances the modularity and the re-usability of the resulting software. Furthermore, the fact that such ideal manager processes know nothing about the tasks performed by the workers they coordinate, makes them generic and re-usable too. The cooperation protocols for a concurrent application can be developed modularly as a set of coordinator processes. It is likely that some of such ideal managers, individually or collectively, may be used in other applications, coordinating very different worker processes, producing very different results; as long as their cooperation follows the same protocol, the same coordinator processes can be used. Modularity and re-usability of the coordinator processes also enhances the re-usability of the resulting software.

410

4. MANIFOLD

In this section, we briefly introduce **MANIFOLD**: a coordination language for managing complex, dynamically changing interconnections among sets of independent, concurrent, cooperating processes [2], which is based on the IWIM model, described in subsection 3.2.

The **MANIFOLD** system consists of a compiler, a run-time system library, a number of utility programs, and libraries of built-in and predefined processes [4]. A **MANIFOLD** application consists of a (potentially very large) number of processes running on a network of heterogeneous hosts; some of which may be parallel systems. Processes in the same application may be written in different programming languages. Some of them may not know anything about **MANIFOLD**, nor the fact that they are cooperating with other processes through **MANIFOLD** in a concurrent application.

The library routines that comprise the interface between MANIFOLD and processes written in other languages (e.g., C), automatically perform the necessary data format conversions when data is routed between various different machines.

4.1. Processes

The atomic workers of the IWIM model are called atomic processes in MANIFOLD. Any operating system-level process can be used as an atomic process in MANIFOLD. However, MANIFOLD also provides a library of functions that can be called from a regular C function running as an atomic process, to support a more appropriate interface between the atomic processes and the MANIFOLD world. Atomic processes can only produce and consume units through their ports, generate and receive events, and compute. In this way, the desired separation of computation and coordination is achieved.

Coordination processes are written in the MANIFOLD language and are called manifolds. The MANIFOLD language is a block-structured, declarative, event driven language. A manifold definition consists of a header and a body. The header of a manifold gives its name, the number and types of its parameters, and the names of its input and output ports. The body of a manifold definition is a block. A block consists of a finite number of states. Each state has a label and a body. The label of a state defines the condition under which a transition to that state is possible. It is an expression that can match observed event occurrences in the event memory of the manifold. The body of a simple state defines the set of actions that are to be performed upon transition to that state. The body of a compound state is either a (nested) block, or a call to a parameterized subprogram known as a *manner* in MANIFOLD. A manner consists of a header and a body. As for the subprograms in other languages, the header of a manner essentially defines its name and the types and the number of its parameters. A manner is either atomic or regular. The body of a regular manner is a block. The body of an atomic manner is a C function that can interface with the MANIFOLD world through the same interface library as for the compliant atomic processes.

4.2. Streams

The asynchronous communication channels in MANIFOLD are called *streams*. A stream has an infinite capacity that is used as a FIFO queue, enabling asynchronous production and consumption of units by its source and sink. The sink of a stream requiring a unit is suspended only if no units are available in the stream. The suspended sink is resumed as soon as the next unit becomes available for its consumption. The source of a stream is never suspended because the infinite buffer capacity of a stream is never filled.

Note that as in the IWIM model, the constructor of a stream between two processes is, in general, a third process. Stream definitions in MANIFOLD are generally additive. This means that a port can simultaneously be connected to many different ports through different streams. The flows of information units in streams are automatically replicated and merged at outgoing and incoming port junctions, as necessary. Thus, a unit placed into a port that is connected to more than one outgoing streams is duplicated automatically, with a separate copy placed into each outgoing stream. Analogously, when a process attempts to fetch a unit from a port that is connected to several incoming streams, it obtains the first unit available in a non-empty incoming stream, selected non-deterministically.

4.3. Events

In MANIFOLD, once an event is generated by a process, it continues with its processing, while the event occurrence propagates through the environment independently. Any receiver process that is interested in such an event occurrence will automatically receive it in its *event memory*. The observed event occurrences in the event memory of a process can be examined and reacted on by this process at its own leisure. The event memory of a process behaves as a set: there can be at most one copy of the occurrence of the same event generated by the same source in the event memory. If an event source repeatedly generates an event faster than an observer reacts on that event occurrence, the event memory of the observer induces an automatic sampling effect: the observer detects only one such event occurrence.

4.4. State transitions

The most important primitive actions in a simple state body are: create and activate processes, generate event occurrences, and connect streams between ports of various processes. Upon transition to a state, the actions specified in its body are performed atomically in some non-deterministic order. Then,

the state becomes *preemptable*: if the conditions for transition to another state are satisfied, the current state is preempted, meaning that all streams that have been constructed are dismantled and a transition to a new state takes place. This event-driven state transition mechanism is the only control mechanism in the MANIFOLD language. More familiar control structures, such as the sequential flow of control represented by the connective ';' (as in Pascal and C), conditional (i.e., 'if') constructs, and loop constructs can be built out of this event mechanism, and are also available in the MANIFOLD language as convenience features.

4.5. Example: Fibonacci series

It is beyond the scope of this paper to present the details of the syntax and semantics of the MANIFOLD language. However, because MANIFOLD is not

very similar to any other well-known language, in this section we present a simple example to illustrate its features and the capabilities. The purpose of the program shown below is to print the Fibonacci series, defined as: $f(1) = 1$, $f(2) = 2$, $f(n) = f(n-1) + f(n-2)$, for $n > 2$.

The first line of this code defines a manifold named `PrintUnits` that takes no arguments, and states (through the keyword `import`) that the real definition of its body is contained in another source file. This defines the 'interface' to a process type definition whose actual 'implementation' is given elsewhere. Whether the actual implementation of this process is an atomic process (e.g., a C function) or it is itself another manifold is indeed irrelevant in this source file. We assume that `PrintUnits` waits to receive units through its standard input port and prints them. When `PrintUnits` detects that there are no incoming streams left connected to its input port and it has done printing the units it has received, it terminates.

In this program, we use two other imported manifolds: `variable` and `sum`. The manifold `variable` reads units from its `input` port, and produces a copy of the most recent received unit whenever a stream is connected to its `output` port. The parameter specifies an initial value.

In the specification of `sum` there is a new linguistic element. In addition to the default ports that all manifolds have, this manifold has two input ports named `x` and `y`. The interface declaration of `sum`, thus, contains the declarations for these ports.

```
manifold PrintUnits() import.
manifold variable(port in) import.
manifold sum(event)
  port in x.
  port in y.
  import.
event overflow.

auto process v0 is variable(0).
auto process v1 is variable(1).
auto process print is PrintUnits.
auto process sigma is sum(overflow).

manifold Main()
{
  begin: (v0 → sigma.x, v1 → sigma.y, v1 → v0, sigma → v1, sigma → print).
  overflow.sigma: halt.
}
```


An instance of `sum` reads a pair of units, one from each of its input ports `x` and `y`, verifies that they contain numeric values, adds them together, and produces the result in a unit on its `output` port. It then tries to obtain a new pair of input units to produce their sum, and continues to do so indefinitely, as long as its input ports are still connected to incoming streams. If a pair of

input values are so large that their addition causes an overflow, `sum` produces a special error unit on its `output` and generates the event `overflow`. Next, is the declaration of `overflow` as an event.

The following four lines define new instances of the manifolds `variables`, `PrintUnits`, and `sum`, and state (through the keyword `auto`) that these process instances are to be automatically activated upon creation, and deactivated upon departure from the scope wherein it is defined; in this case, this is the end of the application. Because the declaration of the process instance `print` appears outside of any blocks in this source file, it is a global process, known by every instance of every manifold whose body is defined in this source file.

The last lines of this code define a manifold named `Main` that takes no parameters. Every manifold definition (and therefore every process instance) always has at least three default ports: `input`, `output`, and `error`. The definition of these ports are not shown in this example, but the ports are defined for `Main` by default. The body of this manifold is a block (enclosed in a pair of braces) and contains only a single state. The name `Main` is indeed special in MANIFOLD: there must be a manifold with that name in every MANIFOLD application and an automatically created instance of this manifold, called `main`, is the first process that is started up in an application. Activation of a manifold instance automatically generates an (internal) occurrence of the special event `begin` in the event memory of that process instance; in this case, `main`. This makes the initial state transition possible: `main` enters its only state—the `begin` state.

Figure 1 shows the connections made among various processes in the `begin` state of `main`. In order to understand how our set of connections produces the Fibonacci series, we consider the sequence of units that flow through each stream. Let α be the sequence of units produced through the `output` port of the process `sigma`. Clearly, this is the sequence of units printed by `print`, and we want to show that it is indeed the Fibonacci series.

The sequence of units that show up at the `input` port of `v1` is, obviously,

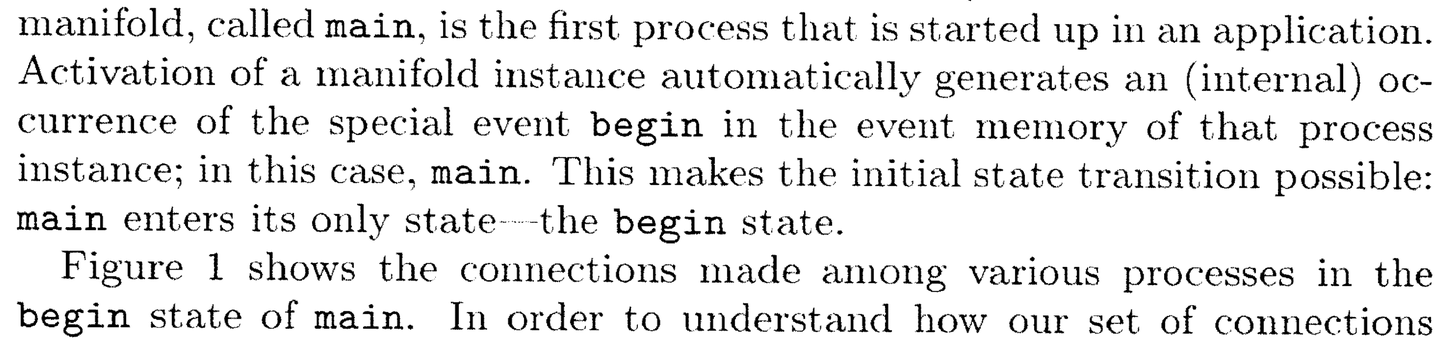

Figure 1. The Fibonacci series.

α. This means that the sequence of units produced through the `output` port of `v1` consists of '1' (the initial value of `v1`) followed by α. This same sequence shows up at the `input` port of `v0` and at the port `y` of `sigma`. It follows that the sequence of units produced through the `output` port of `v0` (which shows up at the `x` port of `sigma`) consists of '0' (the initial value of `v0`), followed by '1', followed by α.

Now, observe that the first pair of units that arrive at the ports `x` and `y` of `sigma` contain respectively, '0' and '1'. Thus, by the definition of `sum` (of which `sigma` is an instance), the first unit in α contains '$0+1$', i.e., '1'. Therefore, the second pair of units that arrive at the ports `x` and `y` of `sigma` contain respectively, '1' and '1' (the first unit in α). Hence, the second unit in α contains '$1+1$', i.e., '2'. The third pair of units that arrive at the ports `x` and `y` of `sigma` contain respectively, '1' (the first unit in α) and '2' (the second unit in α), which produces a '3' for the third unit in α.

This configuration of processes will continue to produce the Fibonacci numbers 1, 2, 3, 5, 8, 13, 21, 34, 55, etc., until `sigma` encounters an overflow. In reaction to the occurrence of the event `overflow` generated by `sigma`, `main` makes a transition to its corresponding state. The transition out of the `begin` state preempts (i.e., breaks up) its stream connections. Both `sigma` and `print` terminate as soon as they detect they have no incoming streams. The transition into the new state executes the action `halt`, which terminates the `main` process.

This simple example shows the power of the 'plumbing paradigm' that is the basis of IWIM and MANIFOLD. What is not demonstrated in this example is the dynamic capabilities of MANIFOLD: that processes can be dynamically created and deleted, and the topology of their interconnecting streams can dynamically change in reaction to the events of interest. Some such examples are presented elsewhere, e.g., in [2, 3].

Separation of computation from communication concerns which is enforced by MANIFOLD leads to separate modules for computation and coordination of communication. This enhances the re-usability of modules, especially, that of the communication modules which are the most complex and time consuming parts of a parallel/distributed application. As a concrete example of this notion of re-usable coordinator modules, it is worth mentioning that a MANIFOLD program written to implement a parallel/distributed bucket sort algorithm was later used, with no change, to implement a single-grid domain decomposition numerical algorithm. It turned out that, although the computations performed in the sort and the domain decomposition problems are very different, the coordination models they required were exactly the same. Furthermore, extension of the domain decomposition problem to the multi-grid case, required only a small change (the addition of a feed-back stream) to this coordinator module.

No modification to any source code is necessary when a MANIFOLD ap-

plication is to run on a single processor machine, a multiprocessor machine, or on a (homogeneous or heterogeneous) network of such machines.

5. CONCLUSION

In this paper, we illustrate the shortcomings of the direct use of communication models that are based on 'Targeted-Send/Receive' primitives in large concurrent applications. We argue that there is an urgent need for practical models and languages wherein various models of cooperation can be built out of simple primitives and structuring constructs.

We present the IWIM model as a suitable basis for control-oriented coordination languages. The significant characteristics of the IWIM model include compositionality, which it inherits from data-flow, anonymous communication, and separation of computation concerns from communication concerns. These characteristics lead to clear advantages in large concurrent applications.

MANIFOLD is a specific coordination language based on the IWIM model. **MANIFOLD** uses the concepts of modern programming languages to describe and manage connections among a set of independent processes. The unique blend of event driven and data driven styles of programming, together with the dynamic connection graph of streams seem to provide a promising paradigm for concurrent systems. The emphasis of **MANIFOLD** is on orchestration of the interactions among a set of autonomous *agents*, each providing a well-defined segregated piece of computation, into an integrated concurrent system for accomplishing a larger task.

The **MANIFOLD** system runs on multiple platforms. Presently, it runs on IBM RS6000, IBM SP1/2, HP, SUNOS, Solaris, and SGI IRIX. Linux and Cray Unicos ports are under way, and other ports are planned. Our present and future work involving **MANIFOLD** includes completion of a visual programming and debugging environment we are developing for **MANIFOLD**, and using **MANIFOLD** in industrial High Performance Computing applications.

REFERENCES

1. D. GELERNTER, N. CARRIERO (1992). Coordination languages and their significance. *Communications of the ACM 35*, 97-107.
2. F. ARBAB, I. HERMAN, P. SPILLING (1993). An overview of manifold and its implementation. *Concurrency: Practice and Experience 5*, 23-70.
3. F. ARBAB (1995). *Coordination of Massively Concurrent Activities*, CWI report CS-R9564, Amsterdam.
4. F. ARBAB (1995). *Manifold Version 2: Language Reference Manual*, Tech. Rep. preliminary version, CWI Amsterdam.