Practical Concurrent Programming for Parallel Machines

D. B. SKILLICORN*
Department of Computing and Information Science, Queen’s University, Kingston, Canada K7L 3N6

Concurrent programming has been viewed as an activity in which a knowledge of the target machine architecture is indispensable to building efficient software. The variety of possible parallel machines has spawned a corresponding variety of concurrent languages for programming them. This has prevented the significant use of parallelism in all but high-end numerical computation because it has not been clear which approach was best.

Much of the confusion could be dispensed with if concurrent programming languages that were independent of the particular target architecture could be found. Despite some scepticism about this possibility, several such languages now exist. We survey them and discuss their merits and drawbacks.

Architecture independent programming languages can be divided into two broad classes: those that are based on the PRAM model or some extension of it; and those which dispense with a need for explicit storage management and scheduling. Languages in the former class provide guarantees of performance but require much of the programmer; those in the latter do not provide performance guarantees but are much less demanding to use.

Received July 1990

1. INTRODUCTION

A wide range of parallel computers are now available and an even larger variety of methods have been proposed for programming them. We argue that concurrent programming languages that apply only to one type of parallel architecture are not of long term usefulness. We survey existing approaches to concurrent programming that are to some extent architecture independent and discuss their relative merits.

There are many different ways of classifying parallel architectures (Ref. 31) depending on which properties are of interest. In this paper we will use four broad divisions which differ in important ways as targets for parallel software development.

The first class are SIMD computers. These machines have a single instruction processor which broadcasts actions to a set of data processors that can execute concurrently. The data processors are connected by a switch and can exchange data synchronously; typically the communication switch can realise an arbitrary permutation of the data. The important property of such machines is that only a single action can occur at each step but it may apply to many distinct data.

The second class are tightly-coupled MIMD computers. These machines consist of a set of processors, usually each with some local memory even if only a cache, connected to a set of memory modules by a switch or bus. The memory modules can be accessed by any processor and therefore permit processors to communicate and synchronise. Distinct actions can take place in each processor simultaneously and, unless the computation has high data locality, references to the shared memory take place on most steps.

The third class are loosely-coupled MIMD computers with a rich interconnection topology. Such machines distribute the entire memory so that each processor owns a memory module. The processor-memory pairs are connected by a (usually static) interconnection topology. No processor can access another processor’s memory except by sending a message to the owning processor. We consider a machine to be in this class if the number of links per processor provided by the interconnection topology is at least logarithmic in the number of processors. We call such machines hypercuboid.

The second and third classes can be regarded as extremes of the same class. The switch of a tightly-coupled machine imposes latency on each access to the shared memory; for typical dynamic switches this latency is logarithmic in the number of processors in the machine. Similarly the diameter of the interconnection network of a loosely-coupled MIMD machine is logarithmic in the number of processors. Thus both classes impose similar costs on remote memory references. The difference between the two classes lies in how remote references can be made less frequent. In purely tightly-coupled machines this is done entirely at run-time by caching frequently accessed values on the processor side of the switch. In purely loosely-coupled machines this is done entirely at compile-time by placing data in the memories of particular processors. A tightly-coupled machine in which caches were pre-loaded would resemble a loosely-coupled machine; a loosely-coupled machine that migrated data as it was referenced would resemble a tightly-coupled machine. Thus the distinction between these two classes depends on the designer’s belief about the best way to get locality of reference; and each class can be merged with the other by scaling the size of local memory relative to global, shared memory.

The fourth class are loosely-coupled computers with a sparse interconnection topology. Such machines have the same processor/memory units as the hypercuboid class but the number of links per processor is constant (independent of the number of processors). Such machines have limited communication bandwidth that significantly affects their performance.

Examples of machines in all four classes are commercially available, and have been for some time. Different architectural properties have led to different techniques in software. SIMD machines encourage data parallelism; tightly-coupled MIMD machines encourage cheap access to shared data; and loosely-coupled MIMD
machines encourage the use of message passing primitives. The MIMD classes must also be concerned with synchronisation: atomic test-and-set instructions for tightly-coupled machines; synchronisation by message reception for loosely-coupled machines.

As a result the paradigms and programming languages that have grown up around each class are significantly different. Even when the programming language used is superficially the same (Fortran perhaps) the primitives added to it because of machine features make it very different to use. Software developed for one kind of machine cannot be moved in any reasonable way onto another kind (and sometimes even a similar one of a different size). Doing so would require recoding it using a new set of synchronisation and communication actions; often the granularity of code segments would also have to be altered. Even the repertoire of standard algorithms and ways of handling common constructs varies between classes. Programmers therefore develop particular skills that restrict them to software development for one class of architecture. The net effect is that software developers and users who migrate to concurrent software development will tend to become locked in to the type of parallel machine that they first use.

This problem would not matter if one of the architecture classes were a clear winner in terms of performance and long term growth. In fact, this is far from the case. Those machines that seem easiest to use today, such as tightly-coupled MIMD machines, are those which will not scale. Such machines today may contain 1000 processors; they will probably not scale more than another decimal order of magnitude. On the other hand, SIMD and sparse MIMD systems can apparently be scaled to very large sizes, but they are harder to program. Even if a user has a particular class of applications in mind, an architecture choice is not obvious because of the lack of a parallel complexity theory that accurately reflects that real behaviour of the underlying system.

It is scarcely surprising that most potential users have avoided making the transition to parallel machines and development of parallel software. Parallel machines have been available for a decade but their market penetration remains small, despite the significant cost/performance benefits that they enjoy over high-end vector machines. Users have chosen to wait, some exploiting parallelism through automatic parallelisation of existing sequential code.

The solution to these problems lies in finding concurrent programming languages or programming models that are architecture independent. They must be able to be efficiently implemented on many different architecture classes (preferably all of them) with little or no loss of performance. It was widely believed until recently that no such concurrent language was possible; that the gap between a programmer’s abstract view of the machine and the various real machines was too large to be efficiently crossed. This is not the case, as we shall see.

This paper presents a survey of realistic ways to develop software that exploits parallel machines; not only the machines now available but also those that can be expected in the future. It is based on theoretical work that has clarified the limits of performance of different types of parallel machines. Despite the apparent plethora of approaches to concurrent software, there are only a few practical solutions.

The paper is organised as follows: in the next section we suggest why the goal of continuing to develop software for sequential systems and the parallelising it at compilation is not a good long term strategy. In Section 3 we introduce the notion of a computation model and discuss the two most popular parallel computation models. In Section 4 we consider the PRAM model and show that it is too powerful to be implemented directly. We also introduce the concept of universality and show that the PRAM model is universal over two parallel architecture classes. In Section 5 we discuss several enhancements of the PRAM model. Restricting the PRAM model can restore universality over the sparse loosely-coupled MIMD class; in Section 6 we see how this can be done by reducing frequency of communication; in Section 7 by reducing the distance over which communication takes place. Sections 8 and 9 discuss models more powerful than the PRAM model in which communication is made explicit and requirements for explicit scheduling are weakened. Section 10 touches briefly on computation models that are architecture specific.

2. WHY NOT PARALLELISE SEQUENTIAL SOFTWARE?

It is natural to ask at this point why software is not developed as it always has been, using languages and software engineering techniques designed for sequential execution. The job of utilising the available parallel hardware can then be delegated to a compiler which extracts parallelism. Programmers can continue to think as they do now, while the extraction of parallelism is done automatically.

This approach has been quite successful in exploiting the power of vector architectures. Compilation techniques are good at finding parts of programs for which vectorisation is appropriate. Some of these techniques will also discover program sections that can be executed concurrently, so that this approach can be used for parallel machines. Parallelising compilers preserve the investment in the huge volume of existing sequential software. It is much less clear that continuing to write sequential software and parallelising it forms a sensible path forward.

Parallelising compiler technology was the result of a large research thrust over fifteen years. It is already showing signs of diminishing returns. It seems intrinsically perverse to have a programmer, who sees something at least of the opportunities for parallel execution of an algorithm, force sequential dependencies onto the code, only to have the compiler try to reconstruct the parallel structure of the program. While compilers are better than humans at some parts of the translation process, notably optimisation, sequentialising programs introduces many spurious dependencies, which are extremely hard to detect as such. Information about opportunities for parallelism has been lost. Anything which makes it possible to use the programmer’s understanding of the computation rather than the compiler’s inferences will allow more parallelism to be exploited. One of the central problems of this decade is
to find the right software framework for parallelism; one in which programmers find it easy to express computations, and which also allows computations to make full use of the available hardware.

3. COMPUTATION MODELS

A model of computation is an abstraction of a physical machine that captures the properties of the machine that are visible at the software level. For the sequential world, the RAM model that lies behind imperative languages is very close to the underlying hardware. This is especially true for CISC architectures such as the NS32000, and less true for RISC and VLIW architectures; but sequential software development has gone on in comparative ignorance of the underlying machine. The apparent complexity of sequential software reflects the actual complexity of executing it—high level language statement counts are proportional to machine instruction counts.

For parallel computation models the situation is very different. If we want a model that can be useful for more than one architecture class, then the correspondence between model and machine cannot be so direct. The computation model must abstract from the substantially different characteristics of different machine types: shared versus distributed memory, same versus different instructions and so on. However, we do still want to have a correspondence between apparent complexity at the software level and actual complexity of execution. Thus developing reasonable computation models for parallel software is harder than for sequential software because the abstraction required is relatively greater.

The two most popular parallel computation models are: the PRAM model, and the circuit model. A PRAM is an abstract machine consisting of a collection of processors, possibly each possessing some local memory, and a large shared memory. In unit time it can: perform a simple computation, read a value from memory, or write a value to memory. Processors can only communicate via the shared memory (one writing and the other reading) and no two processors can attempt to access the same memory location on the same step. This is called an EREW (Exclusive-Read-Exclusive-Write) PRAM. PRAM complexity measures are: the number of steps an algorithm takes; and the number of processors used. For an input of size $n$ we will indicate the time complexity by $t(n)$ and the number of processors by $p(n)$. The memory size is assumed to be unbounded, as it is in the sequential RAM model.

A circuit is a directed acyclic graph whose nodes are labelled with simple operations and whose edges represent the flow of values from one computation to another. Nodes are restricted to bounded fan-in and fan-out. The complexity measures are: time, measured as the length of the longest path between an input and the output of the circuit; and hardware, measured by the number of gates.

Recall that the criteria for a good parallel computation model are that it should be abstract enough to be implemented on different parallel machines and that the cost of the implementation should be directly related to the apparent cost in the model. In the next few sections we consider implementations of the PRAM model.

4. THE PRAM MODEL

The PRAM model provides the programmer with the abstraction of shared memory, but requires each operation to be explicitly scheduled. The control flow must be organised into threads; the sequencing of steps within a thread must be described; and it is the programmer’s responsibility to make sure that simultaneous references by distinct processors to the same location are never made.

When the PRAM approach is implemented on any of the MIMD architectures, problems arise because of the shared memory abstraction. On tightly-coupled MIMD (hereafter tcMIMD) machines, the shared memory is implemented by a set of memory modules connected to the processors by a switch. A reasonable cost electrical implementation of such a switch requires depth logarithmic in the number of processors. Hence the time for each memory reference, counted as unit time in the model, is actually logarithmic in the number of processors used. The total time to execute the PRAM computation therefore increases by a $\log p(n)$ factor. Optical interconnects offer some hope of avoiding this but it is too early to tell.

On hypercuboid architectures, the physical memory is distributed, with portions located close to each processor. We cannot assume any locality about the memory references of a given thread; thus in general memory references will require communicating with processors at the remote edge of the communication topology. The diameter of any reasonable hypercuboid topology is logarithmic in the number of processors, so that each memory reference takes time logarithmic in the number of processors — again the total time for the computation increases by a factor of $\log p(n)$.

For sparse loosely-coupled MIMD systems the memory is also distributed, and remote references still predominate. Perhaps surprisingly, it is still reasonable to assume that the diameter of the communication topology is logarithmic in the number of processors (see the result of Bokhari and Raza\textsuperscript{35}) so that the same order delay occurs.

This shows that a PRAM computation cannot be directly implemented on any of these architectures without increasing the execution time by a $\log p(n)$ factor.

An arbitrary PRAM computation cannot be implemented on an SIMD machine with $p(n) > 1$ without increasing the total execution time, but for a different reason. The variation in the actions taken by the different processors cannot be encoded in the broadcast by the instruction processor without extra steps proportional to $p(n)$.\textsuperscript{32} Thus time complexity cannot be preserved by any parallel implementation of a PRAM computation. The assumptions of the PRAM model are, in a certain sense, too strong to be satisfied by real parallel machines.

The strongest property that can be preserved is the total cost of execution, that is the product of the time and processors required: $p(n) t(n)$. A model is said to be universal over an architecture if it can be emulated in a way that preserves this product. The PRAM model is universal over the tcMIMD and hypercuboid classes, but not over the SIMD and sparse MIMD classes.\textsuperscript{32, 36}

The construction used to show universality is interest-
ing because it uses several randomised algorithms, and it suggests approaches to restoring universality for those architectures for which it does not hold. Consider the trace of a PRAM computation that uses $p(n)$ processors and takes time $t(n)$ (for a problem instance of size $n$). Implement the computation on a tightly-coupled MIMD machine with $p(n)/\log p(n)$ processors, multiplexing each group of $\log p(n)$ virtual processors onto a physical processor by executing the first instruction of the first virtual processor, the first instruction of the second, and so on. The time between execution of successive instructions from the same virtual processor is of the order of $\log p(n)$. Provided memory hashing is used to place the variables in memory, contention in the switch can be made negligible (with high probability independent of the size of the computation). Therefore the latency in the switch can attain the lower bound of size logarithmic in the number of physical processors.

The total time for the computation increases because each processor must execute the steps from log $p(n)$ processors – hence the total time becomes $t(n) \log p(n)$. However, the hardware used is $p(n)/\log p(n)$ so that the product remains $t(n) p(n)$; universality is restored. The requirement that the number of virtual processors be larger than the number of physical processors is called parallel slackness.

The same multiplexing technique can be used to implement PRAM computations on hypercuboid machines. Here the ability of hypercuboid networks to deliver log $p(n)$ permutations in $O(\log p(n))$ time is essential – this can be achieved using two phase randomised routing. The fact that memory references can be regarded as permutations is again a consequence of memory hashing.

On sparse lC MIMD machines, this multiplexing technique can still be applied, but a further delay occurs because of the lack of capacity on the sparser connection topology. Suppose that the number of links/processor is $\alpha$. On each step, a memory reference permutation is generated. It will require use of communication bandwidth for a further log $p(n)$ time steps because some messages will travel the diameter of the network. Thus at any time there are messages from log $p(n)$ previous permutations still in the communication network. But the network can only move $\alpha$ of these at a time. Therefore, an attempt to run the machine at full speed will cause a monotonic growth in the number of outstanding messages. The rate of memory access must be slowed by a factor of log $p(n)/\alpha$ to compensate.

Even multiplexing cannot avoid this problem. Multiplexing as before, the total time for a computation becomes $t(n) \log p(n) \log p(n)/\alpha$ while the hardware requirement is $p(n)/\log p(n)$. Hence universal emulation fails. Details of these constructions can be found in Ref. 36.

PRAM computations fail to be universal over SIMD machines because of the bounded variability that an SIMD machine can display. This still holds even if coding tricks are used, since it depends solely on information flow between the instruction processor and data processors.

What are the implications for the future design of concurrent software? Software that can take advantage of these universal constructions must have logarithmic parallel slackness. This amounts to saying that it is not feasible to actually exploit all of the available parallelism in an algorithm in the sense of reducing the execution time by adding hardware. For implementations on sparse lC MIMD machines a further logarithmic slowdown is unavoidable.

McColl has argued in Refs. 23, 24 that these considerations are not very restrictive, at least for scientific and numerical software. Many such applications are 'embarrassingly parallel' so that the requirement for parallel slackness is trivially satisfied for machine sizes that we can imagine (but see the discussion of macro-dataflow later for some of the pitfalls of this assumption). Sparse lC MIMD machines seem to be the most interesting class for the long term and machines in this class, such as those based on the second generation Inmos Transputer H1 scheduled for 1991, will have two phase randomised routing implemented in hardware. Thus, although message passing introduces a logarithmic delay, the constants involved are very small and can be ignored for systems of foreseeable size. Finally, from a pure complexity point of view the main distinction in parallel algorithm complexity is between the class $NC$ of problems with polylog time algorithms requiring polynomial hardware and those that require polynomial time (supposing them to be different – an important open question). The addition of a logarithmic factor caused by messaging overheads does not change a problem’s status with regard to $NC$. The class $NC$ is somewhat unsatisfactory because of the emphasis it places on reducing execution time, possibly at the expense of impractical amounts of hardware. More practical complexity classes have recently been suggested by Kruskal, Rudolph, and Snir.

The exploitation of parallel slackness is not a new idea, although it has been insufficiently appreciated. The performance of the (now defunct) Denelcor HEP depended precisely on exploiting it. Each processor maintained a queue of status words that preserved the context of a particular thread. Successive executions in the same thread were separated by the execution of an instruction from each of the other threads. This guaranteed sufficient time for the execution pipeline to complete the first instruction before another from the same thread was begun.

It should be pointed out that our implication that certain kinds of access can be achieved in logarithmic time is itself an abstraction. Embedding real objects in three dimensional space means that the Euclidean distance between them must grow as the cube root of the number of objects. Hence the average time for an object to communicate with another cannot be smaller than this cube root bound. Fortunately, all of the results we will use can be altered to this more realistic scenario by replacing logarithmic terms with cube root terms throughout.

5. ENHANCING THE PRAM MODEL

We have observed that, while the PRAM model regards memory access as a unit time operation, real implementations must regard it as a logarithmic time operation. This has motivated several researchers to ask what other operations can be implemented on real machines in logarithmic time, and could therefore sensibly be treated as unit time primitives within an enhanced PRAM
model. These enhanced primitives would have the same status as memory access, so that the resulting computation model would still require explicit scheduling but would have a richer set of memory manipulating primitives.

This approach has really only been used for tightly-coupled MIMD machines, because of the flexibility of the switch. The switch wires connecting processors to a particular memory location have the structure of a tree. If some processing power is added to the internal nodes of this tree then memory references and data can be manipulated in transit. The simplest extension is to merge requests from processors for the same location into single requests and duplicate the corresponding value as it is returned along the reciprocal path. This implements concurrent-read since requests from all processors for the value at a given location result in a single read from that location, rather than a linear number of reads. All processors receive the value in logarithmic (real) time.

Concurrent write can be similarly implemented by requiring internal switch nodes to perform some operation on values destined for the same location in memory. The operation depends on the semantics of the concurrent write. The simplest is to discard one value and transmit the other – this means that an attempt to write values concurrently to the same location results in one of those values (non-deterministically) being written. Another possibility is to combine them using some associative operator (as in the Connection Machine\textsuperscript{14}). This kind of PRAM extension has been implemented by the NYU Ultracomputer and the IBM RP3.

The next most sophisticated enhancement of the PRAM model is the addition of the scan, or parallel prefix, operation.\textsuperscript{22} Given an associative operator $\oplus$ and a list of elements $[a_0, a_1, \ldots, a_n]$, the scan computes the list $[a_0, a_0 \oplus a_1, \ldots, a_0 \oplus a_1 \oplus \cdots \oplus a_n]$.

This operation can be implemented by the switch if the internal nodes are able to compute the operation $\oplus$. This can plausibly be argued if $\oplus$ can be computed in a single pass over the bits of its arguments and the size of the arguments does not exceed the logarithm of the number of processors. The construction and many examples of its usefulness in algorithms can be found in Blelloch’s paper.\textsuperscript{10}

A further enhancement to the PRAM model has been suggested by Ranade.\textsuperscript{30} He describes a new primitive called multiprefix which generalises scan. Suppose that some set of $k$ processors references a variable $A$, the processors are ordered, and $\oplus$ is a binary operation. If the initial value of $A$ is $a$ then the execution of the multiprefix $MP(A, v_i, \oplus)$ by processor $i$ results in it acquiring the value $a \oplus v_i \oplus v_2 \oplus \cdots \oplus v_i$ and the variable $A$ ends up with the value $a \oplus v_1 \oplus \cdots \oplus v_i$. Ranade shows that the multiprefix operation will terminate (with overwhelming probability) in $\log p(n)$ steps. Thus it is no more expensive in time than memory reference. Examples of its use can be found in Ref. 30.

Using these kinds of enhancements as part of a programming language is obviously sensible since, although they are treated as unit time primitives within the model, their real implementation takes longer by a logarithmic factor rather than a logarithmic squared factor. There are several open problems: it is not known (although it seems reasonable) whether these primitives can be implemented in the same time as memory access on other architectures such as hypercuboid machines. It is also not known whether the multiplexing technique used to show universality can be used together with these primitives; that is it is not clear whether enhanced PRAM models are universal even over the tcMIMD class of machines.

6. FREQUENCY BOUNDED COMMUNICATION

The difficulty with achieving universality of the PRAM model on sparse lcMIMD architectures arises because the communication requirements are too strong for the available sparse connections. This led Valiant\textsuperscript{35} to suggest a restriction of the PRAM model called the bulk synchronous or XPRAM model. This model retains the explicit scheduling of the PRAM model but reduces the frequency of non-local reference to memory. Thus the abstract machine consists of $p(n)$ processors, each with limited local memory that can be accessed in unit time, and a shared memory that can be accessed in unit time but only infrequently.

Implementing the XPRAM on a sparse lcMIMD machine can be done using the multiplexing technique already discussed. If the interval between references to global memory is at least as long as $\log p(n)$ then the message traffic is reduced to a level where the sparse communication topology can keep up.\textsuperscript{35} This makes universal emulation on such machines possible. The XPRAM is universal over tcMIMD and hypercuboid architectures as well, since they are more powerful architectures.

An XPRAM can be considered a PRAM with an instruction set of varying granularity. Each sequence of ordinary instructions between a remote memory reference can be regarded as a macro-instruction; and the XPRAM computation becomes a PRAM computation using the macro-instruction set. The problem is that the optimal size of a macro-instruction depends on the size of machine on which the computation is to be executed. The interval between remote references must be chosen to reduce the traffic by a factor that depends on the diameter of the communication topology, which in turns depends on the number of processors. Thus a PRAM computation can only be converted to an XPRAM computation when the target machine size is known. To put it another way, the macro-instruction set depends on the machine size and thus cannot be used directly to write (portable) programs.

There is always the possibility that compiler technology might be made to do the construction of the macro-instructions dynamically at compile-time. This approach works quite well in compiling functional programs to supercombinators,\textsuperscript{39} although the lack of side-effects in that setting presumably makes this an easier problem. It does not seem reasonable to expect programmers to be responsible for macro-instruction generation. Questions about the feasibility of this approach remain.

7. LOCALITY BOUNDED COMMUNICATION

The problem of limited capacity of sparse communication
topologies can also be attacked by reducing the average distance that each reference takes. In our discussion so far we have assumed that memory hashing was used to distribute the references to memory evenly and reduce contention. Memory hashing prevents any use of locality of reference; so we were forced to assume that many references were to locations log \( p(n) \) steps away. However, if we can restrict the PRAM model so that references can always be satisfied by ‘close’ processors then we can avoid the loss of universality for sparse IcMIMD machines and for tcMIMD and hypercuboid machines as well.

One way to do this is to restrict the computation model to a set of second order functionals, each of which encapsulates a particular communication pattern guaranteeing locality. I was able to find such a set of functionals\(^\text{32}\) which turn out to correspond to those of the Bird–Meertens formalism.\(^*\) They include primitives like those of enhanced PRAM models. For example, reduction with an associative binary operation (i.e. the computation, from a list, of the value \( a_0 \oplus a_1 \oplus \ldots \oplus a_{n-1} \)); and parallel prefix can be implemented with constant locality communication.

Because these second order functions come from the Bird–Meertens formalism, other benefits accrue. A rich calculus and set of transformation rules has been developed for the formalism; these can be used in software development in the calculational style and also in reasoning about software.

I was able to show that the operators of the Bird–Meertens theory of lists are universal over the three classes of tcMIMD, hypercuboid, and sparse IcMIMD architectures. Even more surprising, there is sufficient regularity in the second order functions to make them universal over the class of SIMD architectures as well.\(^*\) Thus this particular restriction of the PRAM model is universal over all classes of parallel architectures and can truly be called an architecture-independent computation model. It can be directly regarded as a programming language; I am working towards a compiler for different machine targets.

All of these models have retained the abstraction of a shared memory which is not directly manipulated by the programmer; that is, the programmer is not aware of some memory being distant other than from a performance point of view. And all have required the programmer to explicitly plan the scheduling of every instruction. In the next few sections we examine models that weaken these requirements, so that scheduling and memory both become less important to the programmer.

8. MACRO-DATAFLOW

Macro-dataflow, or multi-threaded computation, is a model in which less must be said about scheduling and synchronisation than in models based on the PRAM. Thus programmers have less to do and software can (theoretically at least) be more abstract. Particular differences between dataflow and the PRAM model are:

- It replaces the abstraction of shared memory by a communication abstraction, but one which is based on names rather than explicit specification of channels. Hence the actual implementation of communication is handled by the compiler and run-time system.

- Scheduling is not explicit but is rather determined by preconditions for each operation – firing rules. In the simplest kind of dataflow this means that arguments must be present before an operation can be scheduled. This can be extended to take into account strictness and operational concerns such as the availability of buffers.

- Communication and synchronisation are treated as the same kind of operation and are implemented by the same mechanism.

- There must be large amounts of parallel slackness in order to provide effective hiding of the latency of inter-transaction transits.

- Memory accesses are treated as two separate actions (split phase). One action initiates the access to memory while a second deals with the memory response. These two actions are treated as decoupled completely so that the latency between the two is less critical (although not to performance).

Dataflow is harder to execute optimally than PRAM computations precisely because there is less information on which to base scheduling. Two conclusions can be drawn from dataflow experience: the overhead of making an enabling and scheduling decision for each operation of the size of an operand of von Neumann instruction is large enough to make fine-grained dataflow systems impractically expensive; and too much parallel slackness can actually hurt the overall execution time of a computation.

Using operations of the size of simple arithmetic instructions requires (at least) matching of arguments and simple scheduling for execution. It destroys all locality of reference for instruction code or arguments. There have been many attempts to design and build dataflow machines that work at this level; almost without exception these projects have turned instead to some form of macro-dataflow, in which simple operations are gathered into clusters (strong components,\(^\text{38}\) jumbo instructions,\(^\text{11, 12}\) macro-instructions,\(^\text{4}\) frames\(^\text{57}\)). This reduces the overhead of enabling and scheduling and also makes it possible to use the fastest available von Neumann technology to execute the macro-instructions (for example Refs 27, 28). Such macro-instructions can share memory locations, allowing matching to take place at fixed locations, create some locality of reference, allow prefetch of instructions, and use pipelines secure in the knowledge that they will not block.

A related approach that does not come out of a dataflow background is Dally’s Parallel Machine Interface.\(^\text{16}\) This is an attempt to provide a standard set of (communication) features that can abstract from a variety of architectures and allow software to be transported between them. The central primitive is a send which transmits a block of data, places it in the memory of the receiving processor and enables it as a piece of executable code, called a segment. Thus communication acquires an active property since it can force some operation to take place in a remote processor. PMI resembles dataflow in merging the communication and enabling functions into a single primitive. As in dataflow, a programmer is not explicitly responsible for scheduling, but communication is more obtrusive since communication primitives appear in the source code. The programmer chooses the granularity of operations.
Both systems rely on the existence of enough enabled operations at each processor to conceal the latency involved in moving data and deciding whether newly arrived data enable new operations. However, too much parallel slackness causes problems. Consider the operations on the critical path of the computation. Each such operation finds itself enabled but waiting behind other enabled instructions from other parts of the computation for access to physical processors. The more other enabled instructions, the longer the critical path takes. Unsurprisingly, this leads to sub-optimal global scheduling, with long ‘tails’ as most of the computation has completed, but a large section of the critical path remains. This is observed in dataflow systems, and has led to several proposals for ‘throttling’, that is slowing the rate at which new pieces of code are enabled.1 A related technique is loop bounding, in which only a fixed number of instantiations of a loop body can be enabled at a time.16 Using throttling can sharply reduce the overall execution time of a computation.

Macro-dataflow is still receiving serious attention as a computation model and so it should. But many difficulties remain.

This problem of long critical paths also occurs in implementing the PRAM model directly when the computation is extremely parallel. Universality is only preserved when the parallel slackness is approximately logarithmic in the size of the machine; if it is larger or smaller then performance will be impacted.

9. COMMUNICATION AND NON-DETERMINISTIC SCHEDULING

An even more abstract class of computation models introduces an element of non-determinism into the computation model. One of these, the Linda model, retains explicit communication; but the others, action systems and UNITY, dispense even with that.

In the Linda model,2 memory is treated as a large shared space, but one which is content-addressable, rather than addressed by label as in the PRAM model. It is called tuple space and its contents are tuples of any type and any size. Tuples are placed in tuple space by an out primitive that behaves like a memory write. However, data are retrieved from tuple space using two primitives, in and read, that find a tuple by matching some of the values provided in the call to values in tuples in tuple space. The remaining values are then read from tuple space. In takes a copy of a tuple, while read is a destructive read from tuple space (that is, it deletes the matching tuple). If more than one tuple matches the template, one is selected non-deterministically. The tuple space may also contain code which can be fetched using an eval primitive.

The non-deterministic way in which tuples are selected means that execution of computations is non-deterministic. Thus this model is more abstract than the one described in the previous section since there is no requirement when placing a tuple space that some other process should consume it. The consuming process may not even execute until some later time, since tuple space is persistent. This has made Linda a huge success in certain application areas such as artificial intelligence.

The implementation of tuple space on real machines presents some difficult problems and necessarily degrades performance. Because tuple space must be implemented by memory, it is subject to the same logarithmic order delays that we have already seen for tightly-coupled and loosely-coupled machines. Further delays are caused by the associative nature of the matching process, which is sufficiently complex that no easy way of using associative memory seems to suffice. Still further delay is caused by the need to make read atomic, requiring election of the winning operation when multiple reads occur concurrently, and deletion of replicated copies in distributed memory systems. Essentially the Linda model provides a higher level computation model, but one in which performance guarantees are not available.

The second kind of more abstract model is based on Dijkstra's guarded command language. Two related models have been proposed. In both a computation is described by a loop of guarded statements. On any iteration of the loop, guards are evaluated and one of the statements whose guard evaluates to true is executed. The loop terminates when no guard is true. Such systems can be executed in parallel whenever multiple guards are true on an iteration and the variables involved in the corresponding statements are disjoint.

UNITY14 was developed primarily to aid in constructing software and it is not yet clear how it might be implemented on parallel machines with reasonable efficiency. Action systems were conceived with parallel execution in mind and a calculus directed towards extracting parallelism has been developed.5 An implementation on a loosely-coupled MIMD system based on the Transputer has also been developed.6 For a restricted class of action systems called chooser-committer action systems, a more or less direct implementation in Occam can be constructed. This implementation does not require the use of any distributed control algorithm except for deadlock detection corresponding to action system termination. Therefore it runs efficiently on loosely-coupled systems. The power of chooser-committer action systems relative to other models we have discussed is not known.

The existence of very abstract (in the sense of not requiring detailed execution description by the programmer) programs which can nevertheless be transformed to efficiently executable systems is very encouraging. It suggests that high level architecture independent models with efficient implementations are not out of reach.

10. ARCHITECTURE DIRECTED COMPUTATION MODELS

Architecture directed models represent the logical extension of attempts to restrict the PRAM model; restricting it in such a way as to exploit particular characteristics of an architecture class. For example, systolic arrays represent a type of architecture that is highly specialized; computations developed for it can execute with high performance, but other computations cannot be executed at all, or only very badly. Nevertheless, some research has been directed towards automatically deriving a systolic form of a computation from some model which is easier for programmers to use.

There are very many concurrent languages and programming models. Those which are designed for weak architecture classes, such as sparse lCIMIMD and
SIMD machines, are of some interest because they can be easily implemented on stronger architectures. Thus they possess some portability, although they may be overly restrictive. For example, the CSP paradigm and particularly the language Occam have become popular models for parallel computation. This is mainly because of the existence of the Transputer to efficiently implement the model. Such computations can be implemented on tcmIMD and hypercuboid systems without much difficulty, provided rapid context switch and some degree of parallel slackness are used. Some of the paradigms developed in this framework are of more general interest, for example Ref. 18.

11. CONCLUSIONS

We have shown that there are only a few possible forms of concurrent software development if architecture independence is considered one of the primary goals. In my view, long-term software development limited to one architecture class is doomed to failure as the leading edge architecture style changes. At present, most potential users of parallel machines are holding back, reluctant to risk their future on the present uncertainty. If past experience can be relied on, the first architecture independent solution found will attract most interest. One of the central research challenges of the next few years is to ensure that the first solution is also the best.

Let us now summarize the benefits and drawbacks of the various approaches. Using the PRAM model means keeping the standard parallel programming model and existing parallel algorithms unchanged. It is universal over two of the four architecture classes, and the two most common. On the other hand, the PRAM model does require programmers to manage both scheduling and memory allocation explicitly; thus parallel computation has more of the flavour of algorithm design than straightforward coding. Despite being available for at least a decade, the PRAM approach has not become the standard for developing parallel software. From the performance point of view, efficient implementation depends on parallel slackness whose amount depends on the size of the target machine. This is not a weakness if the amount of parallelism in the application is extremely large. For sparse tcmIMD architectures a logarithmic order slowdown is unavoidable and no useful implementation on SIMD architectures is possible.

The various enhancements to the PRAM model (that is, adding new primitives) are actually restrictions of the computation model since programmers must use the primitives to gain the benefits. For many algorithms the enhanced PRAM models will provide better execution times. Otherwise, the same considerations apply as for the PRAM model.

Restricting the frequency of communication in a PRAM computation is the bulk synchronous model. This model is universal over three architecture classes, but not over the SIMD class. It is a significant restriction because of the sharp distinction between local and global memory references and the need to control the frequency of the latter. This is made worse because the frequency is a function of the size of the target machine. It is an open question to what extent this process can be automated.

Locality bounded computation is a strong restriction of the programmer's view because parallelism is restricted to a fixed set of primitives that have appropriate communication structure. Nevertheless, experience with the Bird–Meertens formalism suggests that this is not as restrictive to concurrent software development as it might seem. The existence of an associated calculus makes software derivation, transformation, and optimization possible, perhaps with some automated assistance. This model is universal over all four architecture classes and is thus the most architecture independent of the models described. It remains to be seen whether the restriction seems worth the performance benefit.

The other models discussed are more powerful than the PRAM model and, correspondingly, less can be said about their performance. The macro-dataflow approach frees the programmer from explicit scheduling (other than by data dependency) and from memory management (since data are transitory, moving values). It acknowledges the need for parallel slackness; but efficient implementation is a difficult problem. Essentially, the programmer is relieved of the need to provide an explicit, optimal schedule; the run-time system must infer one; but can only approximate the optimal. So ease of programming is traded for poorer performance.

Other models make the programmer's job even easier by introducing an element of non-determinism into scheduling. This makes incremental software development possible, but makes it even harder to say much about performance. Linda systems hide significant performance penalties from the programmer; action systems and UNITY may be even worse. Action systems at least have a refinement calculus to aid in correct program derivation.

There is not a clear winner for an architecture independent approach to concurrent software development. But the number of different options is not large and we can expect further theoretical work to reduce them still further. Meanwhile it is clear that important research problems remain.

REFERENCES

6. R. J. R. Back and K. Sere, Deriving an Occam Implement-
Correspondence

Dear Sir,

On page 444 of volume 33, no. 5 you published an article by Bilodeau and Laguionon on the design and use of assembler routines to extend the facilities of Microsoft Fortran on the P.C. The sample of assembler code in Table 7 is likely to mislead your readers. No one with any experience of mixed-language working or of assembler would be likely to need the article. A naive user would puzzle why the instructions

```
MOV CX, 1
CLD
```

```
LABEL: 
```

```
LOOP LABEL
```

are included. They have no practical effect, and set up and count once through a loop. The code is obviously a rather clumsy modification of a string, as opposed to a single character, move routine from other stylistic oddities.

There is no explanation of the calculation of the character address in the comments, and no explanation of the constant 08000H, which is the base address of the screen memory. There is no mention of ‘memory model’. The format and size of the parameter addresses passed depends on this Fortran compiler option.

Your readers who wish to develop assembler programs would find: Mixed Language Programming Guide (Microsoft, 1987); Advanced MS-DOS, by Ray Duncan (Microsoft, 1986) and Assembly Language Programming for the IBM P.C., by David J. Bradley (Prentice-Hall, 1984) of considerable assistance.

Yours faithfully,

JOHN HENDERSON

Department of Computing,
King’s College London,
University of London,
Strand, London WC2R 2LS

310 THE COMPUTER JOURNAL, VOL. 34, NO. 4, 1991