Asyncrony in distributed parallel computing

C. R. Jesshope, D. B. Barsky, A. B. Bolychevsky, and A. V. Shafarenko

Department of Electronic and Electrical Engineering
University of Surrey
Guildford, GU2 5XH, U.K.

Abstract

Fundamental issues are discussed pertinent to the problem of generic, scalable parallel computing. It is argued that a generic system should be a data-driven, bulk-synchronous one with scalable communications. A data-driven architecture adapted to patterns of massively-parallel computing is described, as well as compiler technology which infers data-distribution patterns as types in an advanced type system.

1 Introduction

Why is parallel computing so dammed difficult? Many will quickly disagree with the implication of this question and in defence point to some application or other which has been run on some parallel computer here or there and claim this to be proof that the question is misleading. Whatever, we still maintain that it is so and that furthermore, until parallel computing can be applied easily and with success to all applications, not only those that are embarrassingly parallel, only then can the difficulties be said to be solved.

What we have seen in the exploitation of parallel computers over the last decade is a massive divergence between the success of an easy application and, to put it bluntly, the disaster of a difficult one. As an aside this is also true for the current generations of scalar microprocessors, which require parallelism to fully exploit. Most success stories in parallel computing belong, of course, to the easy category. To some extent this points to a lack of science in the field. What is it that makes an easy application easy? Why is it that we obtain scalable speed up on an easy application but unmitigated slowdown on a difficult one? Can these questions be quantified? Can those metrics be used in designing not only machines, but compilers and software to exploit them? Of course any single application can be solved if you put enough effort into porting it, but we are interested in more than this, we need to generalise.

Let us not be too pessimistic, we have made progress over the last two decades. We can now confidently say that we fully understand what the problems are. So perhaps we are only looking at another decade before we establish the scientific answers required to bridge that gap and make parallel computing truly general purpose.

1.1 Generality and Asynchrony

Let us consider what generality in this context means and requires of us.

Firstly generality demands a scalable solution to parallelism, but theoretically this is impossible, as in the general case no network or communication system is scalable in both cost and performance. The best we can do is to scale the bisection bandwidth at an additional cost of $\log_2 P$ above our linear scaling in the number of processors P. However, this is a slowly varying function for large P and if we hold sufficient communication bandwidth in reserve then we can provide scalability in an engineering sense. Even so, this is probably the most difficult problem to be solved due to a fundamental divergence between silicon chip processing power and input/output capability, we can say that gates are free but pins are scarce. Optics on the other hand offers the opposite, free pins but scarce gates. Perhaps the only solution here is to resort to opto-electronic solutions which make use of arrays of free-space optical channels but retain the high gate count of CMOS silicon chips. Such technologies are now emerging[6].

Secondly generality demands a high level solution to the problem of programming our computer. This is not a new demand; one of us presented a case in 1982 which called for the adoption of sensible data-parallel constructs to be used in programming parallel computers[7]. However, only in the last few years

has it been generally accepted that message passing as a programming paradigm brings the programmer far too close to the grubby details of the hardware with its attendant loss of generality across applications and targets. The use these higher level languages in turn demands more compiler technology, because with this generality we face difficult (but not insurmountable) questions for which answers must be found within the compiler system rather than when the code is written. There are still more demands as our compiler technology itself demands a scientific foundation on which it be based. It is simply not enough to define a means of propagating user intention to the compiler through an ad-hoc set of compiler directives, such as is found in HPF. Some more theoretical framework is required if we are to generalise successfully[10].

If we now turn to architecture we find the requirement for a high-level programming language makes demands on this too, for it demands the notion of a single linear address space, that very same notion that has made such a success of the von-Neumann uni-processor. Physically shared memory is of course ruled out of the question; it is not scalable. Which leaves us with a distributed memory solution where a single address space is implemented through messages passed around a scalable network, an architectural solution which has been adopted in many recent parallel computers, but not without significant problems. Latency now becomes the primary issue, because we now need solutions to avoid the processor stalling every time a remote (highly latent) fetch is required. Viable solutions to this problem include: data-prefetch as found in the Cray T3D and the Alpha microprocessor it uses; caching with coherency maintained over multiple copies of cached data[11], or dataflow scheduling techniques[2]. Unfortunately only the latter of these can be said to be truly general purpose; irregular and data-dependent access patters can kill the performance of the other two.

So we have considered generality but what about asynchrony? Well, it turns out that just about all of the above demands require asynchrony. At the chip level in designing networks and processors it is advantageous to eliminate any global signals, such as clocks and global controls, but one of the major disadvantages of going asynchronous is the higher pinout demanded. A move to optical interconnect would alleviate this. However, the use of an asynchronous processing mode whereby an activity on a processor is triggered by data delivery rather than control is possible already with the standard VLSI design methodology. The reason why it is not being used extensively is because of the fallacy of the conventional dataflow approach, namely that dataflow architectures are believed to need improvement towards a better performance of the local computation structure (the dataflow pipeline) in less parallel situations, whereas what is really necessary is the account of the highly-correlated nonlocal behaviour that a generic massively-parallel program would induce in the system. In one of the following sections we shall discuss the architecture of a data-driven processor that takes the collective behaviour of the system into account.

At the language level, although we advocate a data-parallel approach we still require asynchrony. The current implementations of data-parallel languages such as that found on the Thinking machines CM5 where synchronisation is forced after every data-parallel operation are inefficient. Applying a bulk-synchronous approach, in which global synchronisations occur infrequently and only when absolutely required has a major impact on compiler and language design and finally as we see in the architectural analysis it is only when we provide a fully asynchronous scheduling of operations or blocks of code that we have a truly general purpose solution. Assuming asynchrony, one can define the properties of computation over a single global step in terms of the attributes of the data used in it, since no control dependencies are imposed and the rest has only trivial operational semantics. Here all we require is a type system powerful enough to represent the relationships between different access methods to data in a distributed system, and besides to do it abstractly, without involving any platform-specific information. We have proposed a type system[10] that makes access to data an issue of symmetry and defines appropriate classifications and subtyping relationships. The whole variety of data-parallel computing primitives appears to be reducible to only a few primitives typed to convert data attributes according to their natural semantics.

The remainder of this paper provides some more detail of the research outlined above, which we are doing in the Computer Systems Group at Surrey University.

2 Data-driven massively-parallel architecture: the processor

The advantages of data-driven computing have been known since the late 60's: latency tolerance and fine-grain parallelism. It may look surprising that the leading parallel architecture is still based on the synchronous RISC processor, despite the 2 decades

of dataflow research. The reason why dataflow units have not made it onto the designer's board are not well-known outside the dataflow community, it is the inertia of rejection rather than rational appreciation of difficulties that makes researchers frown as soon as a new "dataflow project" is announced.

The truth is that conventional dataflow systems work very well in a situation that is not believed to be typical enough and not so well in a more typical situation. Indeed, if an application possesses a high degree of spatial locality, there is little or no benefit in asynchronous computing in general, and dataflow architectures in particular. Most of the time the program would spend without communication, doing some steps of the algorithm that, though independent, can be done in some sequential order, so asynchronous scheduling would be in indifferent equilibrium unguided by data dependencies (since none exists). Asynchronous scheduling, while unnecessary in this situation, would still incur some significant overheads, such as tag processing, matching, instruction fetch, etc.; whereas a cached RISC architecture would be completely at home with such "autonomous" mode of operation being able to exploit all its cacheing potential to the full. As a result, the autonomous part of distributed computation, which is large in most of the large applications (CAD, field theory, etc.), is by an order of magnitude better supported by local, synchronous computing.

Assuming that local computations take the major part of processing time, one is justified in asking why the requirement for excess parallelism (i.e. parallelism of the problem exceeding the size of the machine by a large factor) is necessary at all. Indeed, a RISC processor node can quite successfully handle short-range data dependencies down to plain serial computing, since it usually has a more flexible pipelining structure than a dataflow processor. Hence the inception of multithreaded architectures, which are a hybrid of conventional dataflow and RISC[5, 4].

This constitutes the present direction of research in dataflow architectures and it is leading exactly away from generality, and even asynchrony. A program that relies on detection of locality and static data mapping is a program that requires *pattern-sensitive* translation, an adequate machine topology, and is generally not portable. This is due to the fact that it utilises a single pattern of collective behaviour, namely "weakly nonlocal" computing. The whole implementation process for such a pattern is far from being general at any level from language to architecture, and so a machine optimised to support it is unlikely to support anything else with similar efficiency. And "anything else" includes even weakly nonlocal problems whose behavioural pattern has not been recognised by the compiler!

That is all very well though as long as one believes that every application for which massively-parallel computing may be required is guaranteed to be weakly nonlocal during most of its execution time. One can not realistically expect that, however, since whole classes of applications, such as molecular kinetics, unstructured meshes, intricate finite-element problems, etc. either do not possess any locality at all, or have the sort of locality so complex that it is easier to ignore it altogether than exploit sensibly. The only thing that does indeed predominate in the expanding field of massively-parallel computing is the paradigm of data-parallel programming (by which we mean the set of behavioural patterns characteristic of nonscalar processing in general rather than the SIMD principle of operation in particular). The patterns such applications exhibit are strongly *as well as* weakly nonlocal. There are but a few of them, as follows:

Replication is the process of replicating a data object either preserving its rank (i.e. the number of dimensions) or increasing it. The difference between replication and message broadcast is firstly that the former can be partial and secondly that many such replications can take place at a time (e.g. a vector replicating to become a matrix). The most crucial difference, however, is in that replication can be a logical process responsible for some spatial symmetry, rather than physical proliferation of data, e.g. $\sum_j a_{ij} b_{jk}$ is effectively the process of replication of the matrices into 3d with subsequent reduction. The pattern of replication imposes some strong correlations on the work carried out at different nodes and introduces tremendous fan-out of the results, which is very different from binary matching on which conventional dataflow is based.

Reduction is another pattern of collective behaviour which is the application of an associative, commutative operation to a set of elements of an array. This satisfies well the requirement for binary, nondeterministic matching, but presents a synchronisation problem as counting the elements that have been reduced may be expensive if no special hardware is used (it would require the circulation of a counter token along with the the one carrying the partial result). Loop unfolding may not necessarily reduce counting overhead for reductions

as that would necessitate spurious threading of a group of independent (since parallel reductions are not ordered) iterations, at a similar cost. This problem therefore warrants a specific hardware (or system) solution.

Barrier synchronisation is a particular case of reduction which is special due to its use in bulk-synchronous computing. It originates in the nature of the data-parallel paradigm, owing its existence to the fact that a data-parallel assignment introduces *implicit* and *unresolvable* data dependencies so that assignment to a single element is generally noncommutative with reading any other element of the array, not only the updated one. Barrier synchronisation of an array is a pattern whereby this implicit dependency is made explicit by subordinating any read to the conjunction of the update events and thus making impossible read and write events on the same array at the same logical time.

The implementation of barrier synchronisation as reduction uses the logical AND as a reduction operator, but there is also a broadcast propagating consistently with the reduction, e.g. strictly after it has terminated. One might suggest that a global, rather than distributed solution is preferable, such as open-drain networks etc.; that, however, would be a rather restrictive way in a truly asynchronous system. Indeed, nothing should stop an arbitrary group of processors sending new messages to each other as soon as they have got over the barrier, no matter that the state of the rest of the system may still be indeterminate.

From the data-parallel point of view, there is another speciality in the process. An array can be updated in a statically-known structured manner, e.g. row-wise, in which case there is a regular synchronisation pattern that requires a large number of independent synchroactivities needing a general distributed mechanism.

Tag transformation in the case of massive concurrency is not at all arbitrary. The most common pattern is the one occurring at matching a regularly spaced array with another regularly spaced array with a different spacing. The matching in more than 1 dimension is by index whereas the matching *mechanism* is essentially one that compares addresses; to exploit the one-to-one correspondence between the former and the latter, in the simplest case, the system has to be able to quickly perform the following transformation: given a, b, $s = ai + b$, c and d find the value of $ci + d$. This is to be done for i varying in a large range of integer numbers. Naturally, conventional dataflow architectures provide a means of transforming the tag according to any given rule, but that would involve further tokens circulating in the processor at a prohibitive cost. The processor should therefore support the affine tag transformation pattern directly.

Another behavioural pattern of massive parallelism manifests itself in the *throttling* strategy. Throttling is the term used by the dataflow community to refer to the process of scheduling dataflow programs. To sustain high performance, dataflow systems use the greedy scheduling policy, creating a large number of temporary results, which may exhaust the buffer space of the system. The solution is to keep the number of independent activities in the system limited, hence the name throttling. The implementation of throttling differs from project to project. The Manchester machine has a special centralised unit which enables every active process in the system to proceed by giving it a special enabling token; this is hardly a scalable solution. The MIT project has a strategy called "k-bounded loops" whereby for every repetitive activity, an iteration i is forced to precede iteration $i + k$ even if this does not follow from the data dependencies.

In a massively-parallel situation with collective behaviour, there is explicit parallelism of data, so in fact we have an explicit resource space which can be controlled much more finely. Indeed, any massively parallel activity ends in a distributed synchronisation on an assignment target. The index space of the target is the main source of parallelism: assignments to elements with different indices are independent. Our throttling solution is therefore one of *threading* the element assignments, i.e. making them artificially dependent on each other, so that at any given time exactly k element assignments are in progress. Although similar in appearance, this strategy differs from "k-bounded loops" in that throttling is linked with state update and could in principle be performed in a distributed manner, by the processors owing the respective elements of the target. We are only able to do that because a regular pattern of parallelism is explicit in the program (data-parallel assignment).

2.1 Architecture

It should be clear now that the case of data-driven processor in a massively parallel system is special enough to be different from the conventional dataflow

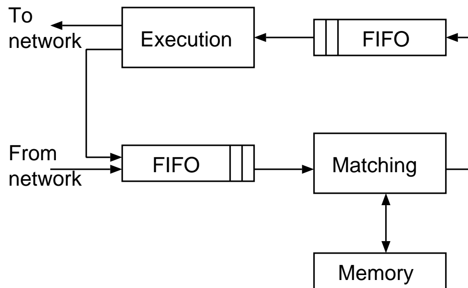

Figure 1: Overall structure of a data-driven processor

solution. In fact, the only thing that is common is the dataflow synchronisation principle and some fundamental architectural features that follow from it: tagged tokens and a matching device. Anything to do with the paradigm of functional programming (which is the natural way dataflow computers are programmed) is irrelevant, as in the massively-parallel situation we deal with large arrays participating in synchronised data-parallel assignments, rather than graph reduction in a stateless paradigm.

All data-driven processors, be them dataflow, multithreaded or otherwise, must follow the overall design presented in fig 1. This is in fact an asynchronous pipeline consisting of the matching and execution units, which need to be equally fast only on average, since for any given token, a memory reference may be required at the stage of matching with a significant slow-down of that stage, but when this is not the case, matching may turn out to be much faster than the actual processing. Another factor which causes imbalance is that tokens are fed to the pipeline one-by-one, whereas an instruction put into effect by the execution unit may cause from zero to two tokens to be produced.

2.2 Matching

The central issue for any data-driven architecture is one of associative memory. Some form of associative memory is required to support matching but it can be (i) real electronic memory with associative access (ii) a simulated associative memory based on a hashing of the key or (iii) direct match memory as in [9]. Each solution has its proponents but none is absolutely suitable for the needs of massively-parallel computation.

The least adequate is solution (i). Associative memory is expensive since it effectively doubles the number of circuit elements required for one machine word. On the other hand, deadlock avoidance requires that for every token to be matched there be a matching cell available on one of the processors. This means that the requirement on the amount of associative memory at a node can hardly be limited and therefore a huge cost may be incurred providing the matching space.

These factors led some of the researchers towards a hashed direct access memory solution, see [3] . While alleviating the problem to a certain extent, this is still a fully associative solution, so what we gain in cost we must lose in speed. The Monsoon project from MIT[9] offered a different matching methodology, whereby a token tag *is* an address identifier (ID), i.e. address of some direct (as opposed to associative) access matching store, rather than abstract key to be matched. Both tokens in a matching pair must have the same ID, with the memory system providing synchronisation on the attempted write. The matching unit of such kind is called Explicit Token Store (the term coined in MIT for Monsoon and used thenceforth with reference to any such device) and it is ordinary rather than associative memory. However every word of it is augmented with some *presence bits* which encode the state of the matching automaton. For example, for the standard binary match there are only two states, and so a single presence bit is necessary. Initially the cell is in the state "empty". As soon as the first token arrives it is written to the cell and the presence bit is set to "full". Once the second token is delivered the attempted write to the same cell (which is full already) will cause a read from the cell and the presence bit returns to "empty" thus allowing for automatic recycling of the ID. The pair of tokens resulted form the match will proceed to the execution unit.

We propose a hybrid solution in which deadlock freedom is based on the provision of ETS matching space for any token circulating in the system. Since ours is the case of massive parallelism the natural choice of ID includes the array element index, which makes the use of ETS preferable to hashing solutions. However, since the structure of computation is statically known, nothing stops us from inducing the initial tokens in some order which minimises the interval of time between matching tokens. This "correlated" way of induction makes far less probable the delivery of tokens to their destinations with a considerable dispersion of tags, although, due to nondeterministic character of the interconnect, no bound on the delivery time may be guaranteed. The optimal solution should combine some fast mechanism of matching tokens with a small difference in the arrival times and the guarantee that in the unlikely event that this difference becomes

large, matching space is still available, though with a slower access. To this end we introduced some real associative space on chip, which handles correlated delivery without using the main memory, and some "drainage" mechanism that ensures that eventually an unmatched token will be removed from the associative memory and placed in the off-chip ETS, where tokens can be stored as long as necessary without causing a deadlock.

2.3 Token format

As distinct from dataflow, we need to decouple the process of matching from execution. The reason why they are not usually kept apart in conventional dataflow architectures is the intended economy of the tag space in a token: since the tag refers to an instruction anyway, it seems reasonable that the information about the firing rule, character of matching and even part of the key may be kept in the instriction. The cost of such a solution is considerable: the instruction needs to be fetched even though the token will not match, purely for finding out the matching cell address and the firing rule, and in a synchronous pipeline this causes irregularities called pipeline bubbles. Also, the main advantage of integrated matching: the fact the key can be broken down into a sizeable "offset", which is kept in the instruction and a relatively small "context" carried by the token, is not effective in the data-parallel case, which can not limit the addresss windows of data objects. Most importantly though, we need more complex matching rules than binary for reductions and replications, for which matching independent of execution is of advantage.

The following information is included in the tag:

- full ID, which points to the location in ETS that the token should use. This is provided no matter whether the associative cache will be effective on the token or not;
- full instruction address, which is used to fetch the instruction *after* the token has met its match;
- site location, pointing to the processor node at which processing will take place (this is used only outside the processor for routing purposes and is neither set nor checked in the processor pipeline);
- control category, which encodes the matching rule, left/right operand, etc.

2.4 Barrier synchronisation

Since barrier synchronisation is important for the case of massive concurrency, we proposed a hardware solution for it which is asynchronous and decentralised (hence, scalable). Any activity can result in the issue of a token with the control category set to special value allocated to barrier synchronisation. Such a token will carry an integer value of one in its data field and a special ETS address in its ID. The matching rule for this control category is binary, with the result token carrying the sum of data values of the operands and the same ID and control category. Before emission, the result token is, however, tested on the data value equal to zero. If that is the case, a new generation of tokens is begun with a different control category and a unary matching rule, which implements a broadcast of the "synchronisation complete" message. Obviously, to make use of this mechanism, all activities participating in the act of synchronisation must issue their synchronising token with the same ID; besides one of them needs to place the negated total number of processes to be synchronised less one in the data field.

2.5 Cross synchronisation

Not all matching in a data-driven system is binary. Replication needs a one-to-many matching as it provides a single source to many recipients. While in a von Neumann system such a pattern would be extremely beneficial (through the use of registers for repeated operands), a data-driven system needs special hardware units in order to avoid (or at least decrease the frequency of) costly physical replication of tokens. In our design of the data-driven processor we have proposed a special synchronisation unit, called "cross-synchroniser", whose intention is to decrease (by a factor of 8) the frequency of replication using a different matching strategy. The device can perform cross-matching of two sets of 8 tokens producing all 64 possible pairs for execution without reference to the ETS. Alternatively, if only one operand is replicated, it can do a 1×64 cross matching, reducing the replication factor by a factor of 64. Both are implemented using a one-word data structure which, for the efficiency of cross synchronisation, should be present in the on-chip associative cache. The logic of the cross-synchroniser is intricate enough to require too much space to describe even briefly. Suffice it to say that it is a 13 stage algorithm which uses a 64-bit-encoded state to decide copies of which tokens should be scheduled for execution and when the whole lot of matching pairs has been produced.

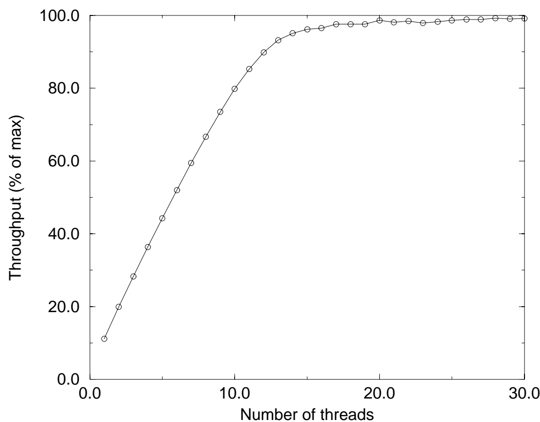

Figure 2: Effective throughput vs number of threads.

3 Networking issues

How is the communication system affected by asynchronous distributed computing? Whereas in the synchronous case the most important parameter is latency, the asynchronous solution can tolerate latency to a large extent, which makes throughput the main factor. We have studied[1] the throughput of a communication system comprising a mesh of MP1 routers [8] which implement the fastest known “mad postman” routing strategy. In order to model “throttling by index”, every processor was assigned a certain number of threads, each being a sequence of exchanges. Every exchange consisted of sending a “request”, i.e. a token, to each of two randomly chosen correspondents on the network and waiting for their responses, i.e. another pair of tokens, to be delivered back to the processor. On receipt of those tokens, the thread attempts to put them onto a synchronous pipeline for “processing” and blocks until this becomes possible. As the pair emerges from the pipeline, the current exchange terminates and a new one begins immediately. Threads are independent, except that they may compete for the first stage of the pipeline, which makes multithreading less efficient.

The purpose of this network simulation was to find out how large the associative buffer would have to be in order for its size to have only negligible effect on performance, assuming that the ETS is very slow. The results we have obtained are summarised in fig 2, where the effective network throughput is shown against the number of threads for a 10 stage pipeline (this is a realistic number of stages in a data-driven system). Note that the amount of fast associative memory required is proportional to the number of threads. The proportion factor is equal to the number of tokens per thread which is typically less than ten. It is clear from the figure that as few as 20 threads exhaust the resources of the system increasing the throughput up to a factor of 7. In reality, much higher gains can be achieved exploiting temporal locality.

4 Programming paradigm

The architecture described above is unable, by itself, to address two other patterns of massively parallel computing, namely uniform data distribution and alignment, which dramatically affect the performance of a massively-parallel computer. In fact, distribution and alignment of data are taken as external characteristics of the program, which are required to be supplied by the user. In this section we shall describe the principles which allow a programming paradigm to convey this information to the compiler.

In our recent paper [10] the problem of automatic classification of massively-parallel objects in terms of their mapping/distribution properties was addressed. A solution was proposed which uses the notion of subtyping in order to qualify the degree of symmetry (understood as independence of an object's properties of certain massively-parallel actions) an object possesses in a symmetry class. Three symmetry classes were considered.

Translational symmetry. Rather than making the fact that the elements of an array do not depend on some of the indices a run-time fact, the comprehensive lattice of translationally symmetric arrays was introduced and type subsumption of more symmetric classes by less symmetric ones was defined. The subtyping rules enable the compiler to unambiguously infer the translational symmetry type of all objects and use the appropriate form of storage for them.

Affine symmetry. Integer arrays with fixed differences between neighbours arise naturally in all index transformations inherent in massively parallel array computing: slicing, extraction of a diagonal, transposition, replication, repacking, as well as any superposition of those. By introducing affine symmetry as a type attribute, these arrays were given the status of first-class objects rather than syntactic features of a programming language, so that the overloaded indexing may be used as the *only* selection primitive. The

compiler is therefore in a position to infer these attributes as types in order to decide how to distribute the data and what hardware means to employ for access. A similar type structure was introduced for affine inequalities, resulting in symmetry types of multidimensional Boolean objects, for which convexity of a spatial domain can be introduced as a type property.

Access symmetry. This introduces classes of objects with respect to specific access patterns. The patterns of total, affine and direct access were considered: total access to an array is selection of all elements of the array but not necessarily in any fixed order, affine access is selection of elements using an affine index and direct access is the selection with just any nonscalar index. Objects were classified into ones intended for use primarily with one or another access type. Since arrays may have more than one axis and the access type is axis-specific, a subtyping lattice was introduced qualifying multidimensional objects in product type.

A system of four high-order primitives was defined and typed in the symmetry-sensitive type system: the application primitive *Map*, the grouping primitive *Juxtapose*, the selection generator *Sel* and the nonscalar concatenation, which along with the known state hiding techniques based on single-threadedness and abstract types is a functionally complete data-parallel programming paradigm which can represent any array computation preserving its parallelism and symmetry. Even data parallelism itself is defined as a symmetry property of application and can be partial as well as complete, to represent execution modes from massively-parallel down to sequential threading.

Finally, the classification and primitives above are so powerful because they are completely data-driven. Even the assignment primitive in this paradigm has a service input, on which it receives the triggering token and a service output on which it signals completion. Asynchrony is *the* factor that makes such flexibility possible (only 4 primitives), since in a synchronous system one would have to consider computing primitives as given ones and globally synchronise after them no matter whether this is motivated by data-dependencies or not. The structure of the object machine, rather than the properties of data objects, would have to be reflected in the programming paradigm making the solution less generic.

5 Conclusions

1. Massively-parallel computing with the data-parallel programming pradigm benefits from asynchrony at all system levels: processor, interconnect and compilation technology.

2. The main benefit to be had as a result of asynchrony in system design is a drastically higher degree of adaptability and hence a greater generality of all means of computing from language down to hardware.

3. We have shown that a truly asynchronous massively-parallel system requires a different data-driven architecture, communication system and compiler technology compared to state-of-the art message-passing multiprocessors and dataflow/multithreaded machines. The design of such a system should address the characteristic patterns of data-parallel collective behaviour.

References

[1] D Barsky. Simulating traffic in data-driven processor networks. Technical Report CSRG95-02, Department of Electronic Engineering, Surrey University, 1995.

[2] A B Bolychevsky. The fundamental issues and construction of a data-parallel dataflow computer. Technical Report CSRG95-01, Department of Electronic Engineering, Surrey University, GU2 5XH, U.K., 1994.

[3] J G D da Silva and I Watson. A pseudo-associative matching store with hardware hashing. *Proc IEE*, 130E/1:19–24, 1983.

[4] V Grafe, J Hoch, G Davidson, V Holmes, D Davenport, and K Steele. The epsilon project. In J L Gaudiot and L Bic, editors, *Advanced Topics in Data-Flow Computing*, pages 175–205. Prentice Hall, 1991.

[5] R A Iannucci. A dataflow/von neumann architecture. LCS Technical Report TR-418, MIT, 1998.

[6] Ieee 1994 proceedings if the first international workshop on massively parallel processing using optical interconnections. IEEE Computer Society Press.

[7] C R Jesshope. Programming with a high degree of parallelism in FORTRAN. *Comp. Phys. Comm.*, 26:237–246, 1982.

[8] C R Jesshope and C Izu. The mp1 network chip and its application to parallel computers. *The Computer Journal*, 36(8):763–777, 1993.

[9] G M Papadopoulos. *Implementation of a general purpose dataflow microprocessor*. Research monographs in parallel and distributed computing. Pitman (London), 1991.

[10] A Shafarenko. Symmetries in parallelism of data. *Computer Journal*, 1995. to be published. Preliminary version available at URL ftp;//orc.ee.surrey.ac.uk/pub/DPar/asd.ps.zip.

[11] N Tomasevic and V Milutinovic, editors. *The Cache Coherence Problem in Shared-Memory Multiprocessors: Hardware Solutions.* IEEE Computer Society Press, 1993.