

# Asynchrony in Parallel Computing - A question of Scale

C. R. Jesshope and A V Shafarenko

Institute of Information Sciences and Technology  
Massey University, New Zealand

&

School of Electronic Engineering, Information Technology and Mathematics  
Surrey University, UK

## 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. We show that data-driven architectures can be achieved without any loss of the efficiency found RISC-based microprocessors. We also show how this data-driven architecture can be adapted to patterns of massively-parallel computing and how this can be supported by compiler technology which infers data-distribution patterns as types in an advanced type system.*

## 1.0 Introduction

This paper will look at asynchrony as used at many different scales in parallel computing, from the chip level, through the instruction set architecture level and finally at the programming level. But first let us ask the question: “why is parallel computing so damned difficult?” Well, I’m sure many of you will quickly disagree with the implication of this question and in defence point to some application or other which has been run at super-speedup on some parallel computer here or there and claim this to be proof that my question is misleading. That may well be so, but the differences here are similar to the differences we find increasingly in society, the differences between the haves and the have-nots. 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 significant 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’s 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. 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.

## 2.0 Generality and Asynchrony

First of all let us consider what generality in the context of general purpose parallel computing and what it requires of us. 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 really are now in a position to say that gates are free but the 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 optoelectronic 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[1-3].

Secondly, generality demand a high level solution to the problem of programming. 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 [6]. However, only in the last few years has it been generally accepted that message passing as a programming paradigm brings the programmer far to close to the grubby details of the hardware, with its attendant loss of generality across applications and targets. To 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 at when the code is written. And 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 [5].

If we now turn to architecture issues 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 in our parallel system, 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 alpha

microprocessor it uses; caching with coherency maintained over multiple copies of cached data[4], or some form of data-flow scheduling techniques[7]. Unfortunately only the latter of these can be said to be truly general purpose; irregular and data-dependent access patterns can kill the performance of the other two. Dataflow however is very inefficient in terms of the instructions executed per application and the pipelines tend to be deep, exacerbating the latency issue.

Solutions to this problem have been attempted in modern RISC microprocessors which use a number of techniques such as speculative and out-of-order execution, which when coupled with multiple outstanding requests to memory can provide a mitigation of the stalling problem. This approach is adopted for the sake of instruction set compatibility but is not optimal in terms of the schedules that it can produce. In effect this is adopting a data-driven approach but that approach is hidden from the compiler by register renaming and small-scale, data-flow scheduling. This approach lacks generality as it adopts a dynamic, peep-hole-like optimisation on instruction scheduling over a limited window in the compiled instruction stream.

Let us now consider asynchrony. Firstly let us try to define what we mean by asynchrony and why the question of scale is so important. In defining systems it is always much simpler to aggregate properties or to describe collective events but in implementation (at least where concurrency exists) it is always more efficient to schedule individual events as soon as any preconditions on their execution have been met. Thus the most efficient specification of a system is a partial order defining those preconditions. A good example of this is the Petri net which is used in the design of small-scale asynchronous circuits and systems. The problem, however, is that such a specification can not capture a large amount of complexity. Some form of information hiding or aggregation is required to manage complexity. In order to resolve this dilemma we have to look at the question of scales and we can define a micro-scale which captures the time scale of the execution of the events we are trying to schedule and a macro-scale which captures the time scale at which variations in execution time or non-optimality in the schedule make little difference to the overall performance of the system. In a synchronous system the micro-scale and macro-scale are the same. In a truly asynchronous system the macro-scale is infinite.

As it turns out, just about all of our above requirements

on the design and implementation of a general-purpose, parallel computers require asynchrony.

Take for example the lowest level of our journey across scale. Here at the circuit level, a synchronous system has a macro-scale defined by the clock period and a micro-scale defined by the gate delays or combined-gate delays of the circuits we are implementing. The driver towards asynchronous circuits in this case is the divergence between local and global time constants and the need to eliminate any global signals, such as clocks and global controls. There are also disadvantages as well and a major one is the higher pin-out demanded by asynchronous circuits. A move to optical interconnect would alleviate this. The use of macro-scale allows globally synchronous/locally asynchronous designs to be achieved providing we can define a macro-scale clock at a suitably low frequency. These techniques have been applied in some recent work on network design described in section 3.4.

At the other end of our journey across scale the macro/micro scale approach is used effectively at the language level. We advocate a data-parallel approach but we still require asynchrony. Current implementations of data-parallel languages such as that found on the Thinking machines CM5 force synchronisation after every data-parallel operation and 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. 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. Thus, here the macro-scale is defined by the bulk-synchronisations within which we have asynchronously scheduled machine instructions and the micro-scale the instruction or groups of instructions which are scheduled synchronously. This brings us to the architectural level. We have in the past considered a pure data-flow architectures to solve this scheduling problem and we have found efficient solutions[7]. However, the economies of scale dictate that the processor solution be as general as possible, by generality here, of course, we mean something entirely different. The processor must be useable in systems from games consoles through to supercomputers! Because of this we have since discarded the pure dataflow approach in favour of a micro-threaded RISC architecture which provides the benefits of data-driven scheduling but does in within a framework of a standard RISC pipeline with

very little change to the base instruction set architecture[8]. This is described in detail in section 3.2.

## 3.0 Data-driven Massively-parallel Architecture

### 3.1 Issues

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 research in the areas of dataflow and subsequently, multi-threaded processors. 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. However, recent advances in multi-threaded processors[8,12] have brought the benefits of data-driven scheduling, while keeping the largely familiar instruction set of the RISC processor. Thus we gain data-driven scheduling with low scheduling overheads in a standard RISC processor.

In programming multithreaded RISC architectures we can use similar techniques to those adopted in dataflow research but in doing so we have the flexibility to vary the granularity of the unit of work scheduled. In the limiting case we can schedule a single instruction as a thread which provides a pure dataflow solution, although probably rather inefficiently. Alternatively we can ignore threading altogether and provide the static scheduling of a conventional RISC architecture. The granularity we adopt in programming a multi-threaded RISC architecture depends on the overheads for creating and synchronising threads. Again we are back to the issues of scale.

In this paper we describe work which originated in pure dataflow and then migrated to a multi-threaded implementation solution. The issues are the same with the exception of the parameter defining the scale of the schedulable unit. As will be seen the later however, this can still be close to a pure dataflow solution, because of the low overhead in our micro-threaded RISC scheduling mechanisms, which can now be reduced to 0,1 or 2 clock cycles per asynchronous thread, depending on

implementation tradeoffs[8].

In general conventional data-driven 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 overheads, whereas a cached RISC architecture would be completely at home with such "autonomous" mode of operation being able to exploit all its caching potential to the full. As a result, the autonomous part of distributed computation, found in regular applications such as CAD and field theory, etc. is better supported by local, synchronous computing. With a micro-threaded RISC based solution however, either option can be adopted and we gain the best of both worlds!

Assuming that local computations take the major part of the 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 through the use of speculative and out-of-order execution. Indeed this constitutes the present direction of research in modern processor design 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 a *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 or can not be detected at run-time through the mechanisms of register renaming.

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 it 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. matrix multiplication 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.

**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 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.

**Data matching** 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(system) 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).

The  $k$  of course is dependent on the matching space available in the asynchronously scheduled processor. In the detail below, this is matched to the size of the register set, where synchronisation occurs.

### 3.2 A dynamically scheduled RISC-based architecture

It should be clear by now that the case of data-driven processor in a massively parallel system is special enough to be different from the conventional dataflow solution. In fact, the only thing that is common is the dataflow

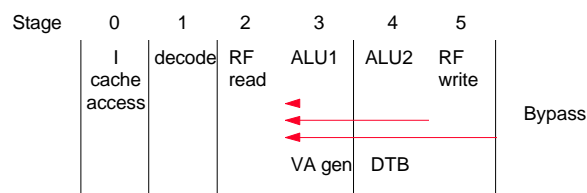


Figure 1 Generic RISC pipeline structure

synchronisation principle and some fundamental architectural features that follow from it. 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.

Let us now consider the architecture described in [8] in more detail. The goal of that work was to design micro-threading primitives that might be added to any modern RISC based processor. Where the resulting design should be completely compatible with existing compiled codes, and which should execute with no loss of efficiency. Moreover, this should require as few additions as possible to the instruction set of whatever RISC design it is based. The techniques we proposed provide very rapid context switching of lightweight threads (micro-threads) which share a single set of registers and the stack. Although our motivation is to support the efficient execution of data-parallel languages, due to the economics of the microprocessor market, the processor must execute existing codes with unaffected performance and should provide enhanced performance even if 'sequential' code is recompiled for it, using established techniques such as the exploitation of functional parallelism in expressions independent statements, loop unfolding, etc.

Let us consider a generic RISC pipeline, illustrated in Fig. 1, which is typical of many current RISC processors. This example has 6 stages with bypass buses from the final three. The first issue which must be considered is how synchronisation may be achieved on the non-deterministic events whose latency must be tolerated.

On RISC processors which are based on the principle of decoupling memory accesses and processing, by adopting a load/store philosophy, it seems an obvious extension to base any synchronisation on additional state associated with the registers, effectively providing a split-phase

asynchronous LOAD mechanism.

An asynchronous LOAD instruction can be implemented by simply providing an additional status bit on each register, which indicates whether the register contains valid data or not. Considering our generic RISC pipeline (Fig. 1), a LOAD is executed as an instruction which normally writes a 'data invalid' state into its destination register. Meanwhile, the data cache read begins at stage 3 and, in the case of cache hit, the fetched data is multiplexed into the output of stage 4 overloading the previously prepared 'invalid' data. If the access misses the cache, then the 'invalid' data state is written into the destination register. The register will be re-written later by the cache memory subsystem (by inserting a bubble into the pipeline or via a dedicated port to the register set). Thus any outstanding memory request must be tagged by the register number into which the data will be written.

We say that a memory access is split-phase if it misses the first level cache. More precisely, a split-phase LOAD is one that can not be satisfied during the life cycle of that LOAD in the pipeline.

Any instruction which reads an 'invalid data' state from either of its operands stalls at pipeline stage 2. It can either wait for the data bypassed from the next stages or merely keep reading the register file. However, in either case, no further processing may take place until the requested data becomes available. Clearly this situation is undesirable, for although the compiler may be able to insert sufficient slack to tolerate the load, any dispersion of latency will destroy the schedule.

To remove this restriction and to perform true dynamic scheduling of several instruction streams in this generic RISC pipeline, we have to introduce the explicit notion of independent points of control, i.e. the manipulation of multiple program counters (PCs) by the processor. Here, a PC represents the minimum possible context information that we can keep for a given thread and in the architecture suggested it is the only reference to a thread. Since several threads can be active simultaneously, some explicit storage for their PCs, called the continuation queue, must be provided. This is associated with the instruction fetch logic at the entry of the pipeline (Fig. 2).

In a normal RISC pipe the next address is transferred from the first stage of the pipe in order to allow the next instruction to follow without delay. Of course, on branch instructions, this normally involves an element of speculation as the direction taken must be predicted and if this prediction fails any subsequent change of state must be 'cleaned up'. We will call this conventional mechanism of transferring control *horizontal transfer* and the alternative mechanism that we propose here, which acts through the continuation queue, *vertical transfer*.

Any instruction can transfer control vertically, horizontally, both vertically and horizontally or indeed not at all. Thus we already have a mechanism for the creation and termination of threads. Of course whenever an instruction does not transfer control horizontally, the next PC is taken from the continuation queue, providing that it is not empty. This mechanism, combined with a modified form of synchronisation described below provides the basic mechanism for latency tolerance.

A thread is created when an instruction which is encoded to transfer control both horizontally and vertically and a thread is terminated when an instruction is encoded not to transfer control at all. How this encoding is achieved is a matter of detailed design but there are two obvious extremes. The first, if the instruction set allows it, is to use two spare bits in some or all instructions in order to encode the direction of transfer, thus allowing any instruction to create or terminate a thread. The other extreme is to add a pair of instructions to the instruction set in order to perform thread creation and termination explicitly, and then to provide a fixed encoding over the remaining instructions in the instruction set to determine whether the instruction transfers control horizontally or vertically. The former solution is more general and thus preferable but should the two bits required not be available, the latter, which decodes the transfer strategy from the an existing instruction code can be adopted. It is not however, always optimal. An instruction which is responsible for a non-deterministic delay (e.g. a LOAD) is not necessarily the one that needs to be coded for vertical transfer; it may be beneficial to code its consumer (i.e. the instruction that reads the register loaded) so that the compiler may insert a sequence of statically scheduled instructions between the two, thus maximising the concurrency available while still minimising the probability of the consumer being put to sleep.

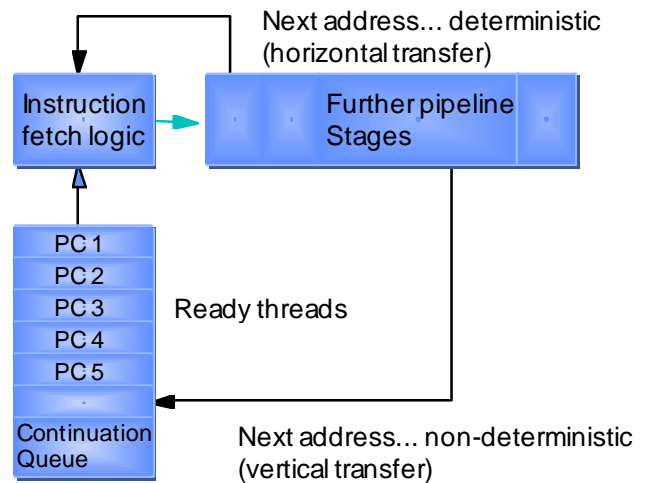


Figure 2. Continuation queue and transfer of control in a micro-threaded RISC architecture

With this scheduling mechanism it is no longer necessary to predict branch direction providing that enough parallelism is available in the code. However, in order to simplify the misprediction recovery, it makes sense to issue the vertical PC from a deeper pipeline stage at which the actual branch direction is already known. If the precise arithmetic exceptions are not required, this can be done right after the register file read (at stage 3 in our example in Fig. 1).

Now that we have introduced a mechanism for dynamic scheduling, it is necessary to revisit the synchronisation mechanism, involving the split-phase load, described above. Any instruction which is dependent on a non-deterministic event, such as access to a register previously loaded from external memory or conditional branch, will normally be encoded to transfer control vertically, pulling another thread into the pipe behind it. Now at the register file read stage (stage 3 in Fig. 1) the instruction either completes and transfers vertically to the continuation queue where the thread waits to be scheduled again or, if the data is not present (or some event has not occurred) the PC itself is written into the register read and that thread is neatly put to sleep.

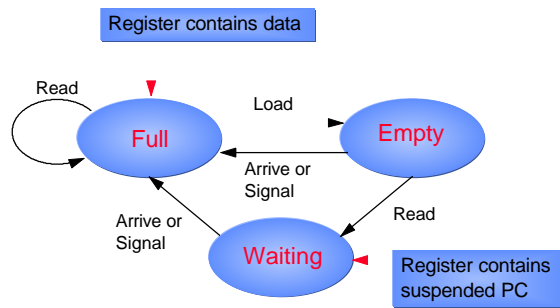


Figure 3. State transitions on a micro-threaded RISC processor's register set.

Figure 3 illustrates the state transitions involved. Clearly all registers are now required to be tagged with two status bits indicating three possible states:

1. The register contains invalid data. The register can be set into this state by a special instruction or a LOAD instruction which misses the primary cache.
2. The register contains a program counter. The register contains the PC of a sleeping thread.
3. The register contains valid data. Any other write of a data sets the register into this state.

The fourth combination of the status bits can be used for a postponed error report.

The way synchronisation is achieved is that when an instruction attempts to read an operand containing

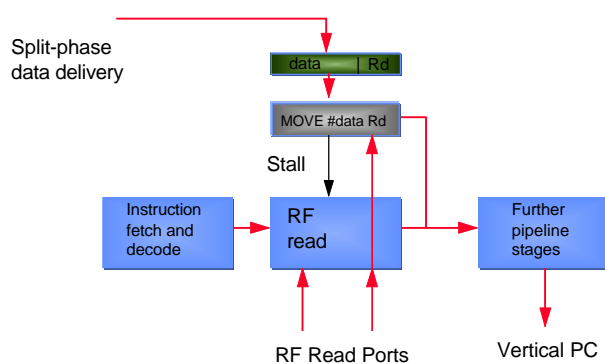


Figure 4. Split-phase load data delivery.

invalid data, the instruction is aborted and transformed

into a 'store program counter' instruction which replaces the register's contents by the PC pointing to that instruction. Hence the thread is put asleep and its PC is kept in the register to which data is expected to be written later. The arrival of that data causes the PC to be pushed out of the register and put into the continuation queue. When rescheduled, the same instruction will find that data in the register its PC just vacated.

These two synchronisation actions (sleep and wakeup) require mutually exclusive read-modify-write access to the register bank. This can be implemented using dynamic dependency control, stall and bypass logic which is embedded in most RISC pipelines.

In executing non-threaded code, all instructions would transfer control horizontally (as there are no additional threads to be executed). This situation is very similar to that of a conventional pipe, a prediction is made (in the case of a LOAD dependency the prediction is that the data is in the primary cache) and the next instruction is executed. Again if an operand contains invalid data a misprediction recovery is performed, i.e. the continuous chain of horizontally fetched instructions headed by the mispredicted instruction is cancelled. However, when there are no other ready threads available it is preferable to use the stall mechanism rather than the sleep-wakeup mechanism to reduce the penalty involved.

In threaded code, if both operands of a dyadic operation are invalid, we can chose either of them as a target for the thread's PC. However, a fixed rule (e.g. always the first) can allow the compiler to perform additional optimisation.

The wakeup action is performed explicitly by means of a special 'move synchronising' instruction which is inserted into a slot created in the pipeline by stalling earlier stages. This instruction moves one register to another. However, it reads both the source and the destination registers and if the destination contains a PC, that PC is issued vertically later in the pipeline. In fact, any instruction reading only one register (e.g. an operation with literal) can be used for this purpose provided it does not itself transfer control vertically.

The manner in which a split-phased memory request delivers the data using a 'move synchronising' instruction is illustrated in Figure 4.

We have shown in the sections above how fork, kill, wait

and signalling may be implemented in a conventional RISC pipeline using its register set as the synchronisation resource. We argued above that in order for micro-threading to be viable it must be possible to initiate and synchronise threads of a few instructions with no significant overhead. In this section we analyse the overheads involved in these various actions.

**Thread creation:** the overhead of a fork depends on detailed design at the instruction set level. Any instruction which has space for an additional address may be encoded to transfer both horizontally and vertically and hence yield a fork at zero cost. Alternatively a variant of a branch instruction may be encoded as a fork. There is also a trade-off in implementation, as it may be necessary (for example in a conditional fork) to transfer control vertically on both continuations and without additional hardware support this could require two additional instructions. The overhead for a fork therefore, is 0, 1 or 2 cycles, known statically, depending on instruction set encoding and hardware tradeoff. The frequency of finding a double vertical transfer in compiled code will determine whether such a tradeoff should be made in order to keep the overhead bounded by a single cycle.

**Thread termination:** the overhead for a kill also depends on detailed design but as no transfer of control is required the overhead is 0 or 1 cycle. Again this will depend on whether a special instruction is added or existing instructions are encoded for transfer of control (in this case the encoding is for 'no transfer').

**Sleep:** this action is implicit in any instruction that reads a register. Storing the PC does not require an additional cycle, as the result of the current instruction is not written if its thread is suspended. However, the suspended instruction must be reissued once the dependency which put it to sleep is resolved. The overhead for a sleep instruction is therefore 0 or 1 cycle, which is not known statically, for it will depend on whether the register contains 'invalid data' or not. More importantly, the overhead for a valid prediction is 0 cycles.

**Wake up:** a wakeup signal may be generated internally or externally, as is the case with a split-phase load. An internal signal may again require a separate instruction, although it can be encoded as an option on any instruction which reads no more than one register and does not transfer control vertically. This latter condition

will often be satisfied as signalling is often performed as the last action of a thread, when it will transfer neither horizontally nor vertically. Thus for an internal signal the overhead is 0 or 1 cycle, known statically. Again there is a hardware tradeoff in the case where no additional instruction is added, as by providing an additional register port the restriction of reading no more than one register may be removed, thus completely eliminating the overhead of internal signalling.

An external signal, such as split-phase load, must create a slot in the pipeline in order to insert the 'move synchronising' instruction. Thus the overhead here is always 1 cycle.

In summary it is clear from the above analysis that micro-threading may be achieved with little overhead. In the case where existing instructions are overloaded with transfer method, the overheads of fork, kill, internal signals and a successfully predicted wait can be eliminated entirely. In the case where additional instructions must be added to an existing instruction set, the overhead for all threading operations may be limited to a single additional cycle.

### 3.3 Processor Analysis

The question we must ask ourselves now is how many threads are required to tolerate the latency found in typical memory systems, as this will determine whether micro-threading is a viable architectural model. In our previous paper[8] we presented an analytical analysis of

node is connected to each of the  $K$  adjacent nodes by a pair of channels: input and output. Routers have two queues for each output channel: one for transit messages and another for messages injected by the processor connected to the router. The transit queue is granted the higher priority, that is the internal queue is served only when the transit queue is empty (in fact, relative priorities of transit and locally-injected messages make

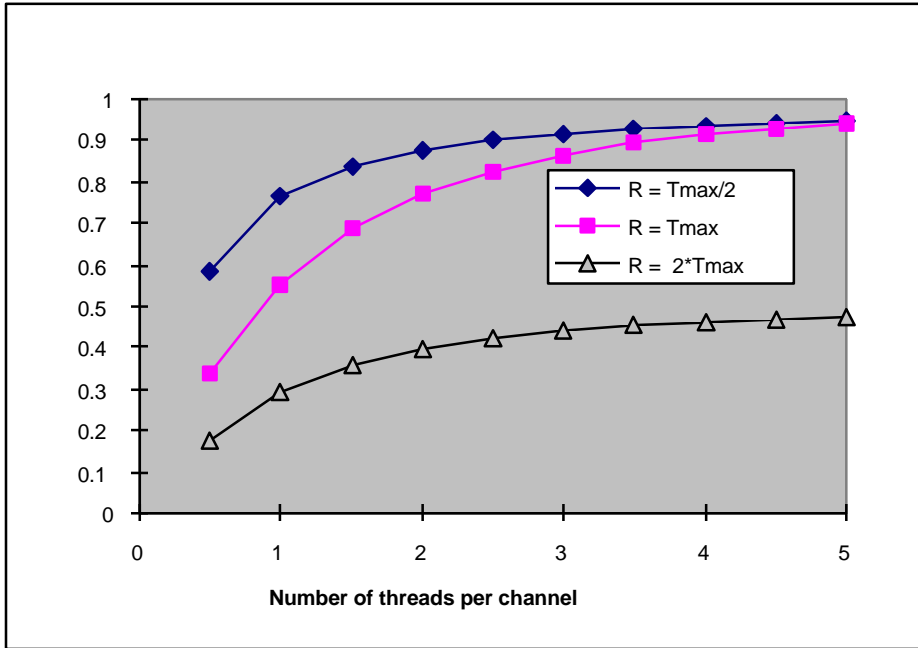


Figure 5. Normalised performance of a micro-threaded RISC processor against the number of threads per channel of the network used to construct a distributed memory system.

the performance of a micro-threaded pipeline as a function of the number of threads. In that analysis we showed that, contrary to the common belief, latency of the memory system is not the only factor that influences this function. Latency has to be considered in combination with other factors including the program behaviour. We modelled two systems, a conventional memory system and a networked memory system. However, in the context of this paper only the latter is of significance.

In analysing the network, we concentrated only on the fundamental features of the network structure itself, not taking into account such factors as contemporary technology constrains etc. In our model, the network is synchronous and messages make one hop in one network cycle. We assume that the network is direct and that each

very little difference for the final result, but the above convention simplifies the calculations). The network is presumed to be loaded by random uniform traffic, and the routing strategy comprises a random choice between the channels prefixing any shortest path to the target.

Figure 5 gives the results of this analysis. It shows a graph of normalised performance versus number of threads per channel in the network providing the distributed shared memory. The number of threads is normalised by the number of channels as in the analysis the number of threads required to achieve a given level of performance does depend primarily on the arity of the network or the number of channels in the router.

For very small networks the graphs for performance,  $P(n)$ , deviate from those shown in Figure 5. due to the impact of lower order terms in the analysis. However this is of very little interest in building scalable parallel computers systems.

The parameter  $T_{max}$  in the graphs represents the maximum throughput of the memory system and the three curves are drawn for various request rates,  $R$ . For  $R = 2 T_{max}$  it can be seen that normalised performance asymptotes to 1, as expected. Moreover what is most interesting is that the number of threads per channel to obtain 80% of the maximum performance is only between 1 and 3. To put this in context, for a 1024 node toroidal network this is equivalent to between 3 and 12 threads, while for a 1024 node hypercube this would be between 10 and 30 threads, in both cases quite modest

numbers. For the third curve ( $R = 2 * T_{max}$ ) however, it can be seen that because the request rate exceeds the maximum throughput of the network-based memory, there will be a penalty in performance. 80% of the maximum obtainable performance however, is still obtained with less than 3 threads per channel.

The conclusion that can be made from these results is that using heavy-weight multithreading for tolerating latency is not justified. The limited number of threads that is sufficient for fairly efficient execution should apparently be drawn from one referential environment with common registers and common stack, which will provide for low-overhead management of threads.

Similar conclusion was made by Culler in [9]. However, his analysis of the maximum number of outstanding requests is based on the technological limitations of existing network-based memory systems. Our 80% at 2 threads per channel figure has nothing to do with those limitations. It is, in fact, a fundamental implication of the fact that the majority of random uniform traffic is in transit in the network

### 3.4 Scalable Networks

As we have already discussed, it is impossible to provide a truly scalable network for a general-purpose parallel computer but that in practice some form of limited scalability may be obtained by using the slowly varying properties of a hypercube. In hypercube networks the cross-section bandwidth can be made to scale but only at a cost which grows logarithmically with the number of nodes. In practice a network maximum size would be chosen which determines the maximum degree of the network and then the engineering solution of a scalability up to the maximum machine size is obtained.

The problem we face here though is in the design of the hypercube switching node itself. Pinout becomes a problem, as does the scalability of the switch node. The latter manifests itself due to the square-law growth of the crossbar required for the switching node. We have already proposed hybrid opto-electronic solutions to the former[3] and recent work at Surrey University[10] aims to overcome the latter.

In this work an alternative, asynchronous solution to the use of a crossbar switch is proposed, based on a very fast ring of registers distributed to the pin or opto-electronic

I/O points on the chip. This acts as a core for the hypercube router. The ring is implemented asynchronously and because the interactions are only local, very high speed circulation of the ring is achieved. Initial simulations in a conservative, 0.7 $\mu$ m CMOS process show a ring cycle of equivalent to 250Mhz. Previous work, based on a mesh router which used a cross-bar architecture, only achieved a cycle time of equivalent to 50Mhz. This approximate comparison gives an indication of the penalty paid for having global connections within a router architecture.

## 4.0 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[5] 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 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:** Access symmetry 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.

## 5.0 Conclusions

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

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.

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.

## 6.0 Acknowledgments

We would like to acknowledge the contribution to the work described in this paper by Alexander Bolychevsky and Ian Swarbrick, both from the University of Surrey. We would also like to acknowledge funding from various grants which have advanced this general area of knowledge from both the EPSRC and European Community.

## 7.0 Bibliography

- [1] IEEE 1995 *Proceedings of the Second International Workshop on Massively Parallel Processing using Optical Interconnections*, IEEE Computer Society Press, 0-8186-7101-7
- [2] Jesshope, C.R. and Swarbrick, I., 1997, The Design and Implementation of a Fast, Low-power, Asynchronous Router, *Proc 4th Annual Australasian Conference on Parallel and Real-Time Systems*, pp114-125
- [3] J A B Dines, J F Snowdon, M P Y Desmulliez, D B Barsky, A V Shafarenko, and C R Jesshope, 1997 Optical interconnectivity in a scalable data-parallel system, *Journal of Parallel and distributed Computing*, **41**, pp120-130

- [4] C R Jesshope 1982 Programming with a high degree of parallelism in FORTRAN, *Comp. Phys. Comm.*, **26**, pp237-246.
- [5] A V Shafarenko 1995 Symmetries in data parallelism *Computer Journal*, **38**, pp365-378.
- [6] M Tomasevic and V Milutinovic (Eds.) 1993 The Cache Coherence Problem in *Shared-Memory Multiprocessors: Hardware Solutions*, IEEE Computer Society Press, ISBN 0-8186-4092-8.
- [7] A Bolychevsky 1994 *The fundamental Issues and Construction of a Data-parallel Data-flow computer*, Technical Report. Computer Systems Research Group, University of Surrey.
- [8] A Bolychevsky, C R Jesshope and V B Muchnick, (1996) Dynamic scheduling in RISC architectures, *IEE Trans. Computers and Digital Techniques* ,**143**, pp309-317.
- [9] Culler, D.E., Multithreading: Fundamental Limits, Potential Gains, and Alternatives, in: Iannucci, R.A., Gao, G.R., Halstead, R.H. and Smith, B. (eds.), *Multithreaded Computer Architecture: A Summary of the State of the Art*, Kluwer Academic Publishers, 1994, pp. 97-138.
- [10] I A Swarbrick and A, Bolychevsky, (1998) Scalable packet router for large binary hypercube networks using optical interconnect, Submitted to Europar '98.
- [12] Iannucci, R.A., Gao, G.R., Halstead, R.H. and Smith, B. (eds.), *Multithreaded Computer Architecture: A Summary of the State of the Art*, Kluwer Academic Publishers, 1994, pp.