Parallel Hardware and Parallel Software: a Reconciliation

P.H.Welch
Computing Laboratory
University of Kent at Canterbury, CT2 7NF, England

Abstract. Parallel hardware is commercially available today at all levels of granularity – from transistors through to vector processing MPPs. Most of this is user-programmable, also at most levels of parallel granularity. A serious problem is that in order to get economically worthwhile performance from these systems, proprietary software needs to be employed to exploit efficiently the underlying parallel architecture. The software models provided to the applications engineer tend to be low-level and specific to each architecture, causing difficulties for secure and verifiable design, long learning-curves and severe portability problems. Recent standardisation efforts (for message-passing interfaces and data-parallel paradigms) still appear to be too low-level and still have to prove themselves efficient.

We need a high-level model of parallelism that is simple and architecturally neutral (so that it can be efficiently supported by current and future hardware platforms) and mathematically sound (so that it can be be made secure by a rich set of analysis and transformation tools). The applications engineer must be shielded from the parallel nature of whatever hardware is to execute the system, but must be exposed to the parallel nature of the application itself and must be able to express that freely with parallel software. Depending upon the application, the parallel software may have no direct reflection in the parallel hardware. Allowing and reconciling this divide is one of the grand challenges facing parallel hardware and software architects.

1 The Pursuit of Performance

The usual basis on which parallel processing is sold is its potential for improving performance. No matter how powerful a single computing node becomes, the prospect of turning around a particular problem in a tenth of the time (or solving a ten-times more ambitious problem in the same time) will always tempt us to investigate harnessing ten or so nodes in parallel.

The problems that arise whenever we yield to this temptation are the same as those involved in the management of any collaborative venture: how do we allocate useful work to all the partners and how do we keep the overheads of coordinating everyone's efforts sufficiently low so that the partnership remains cost-effective? With a few modest exceptions, our experiences with parallel computing have not been wonderful and we remain some way from being able to offer it as a technology to improve the solution of general problems.

If our only motivation for parallelism were to increase the performance of computing systems, we should stop trying to explain its concepts to applications engineers. Parallelism, if we can iron out its private problems, becomes just another technological fix and the user should not need to know it is there. Programming languages should insulate the programmer from the intricacies of parallelism, just as they hide other details of target hardware architecture (like the number and type of registers, levels of cache hierarchy, superscalar pipelines, ...).

There are two problems with the above strategy: one is that its chances for success are low and the other is that the consequences, if it were to succeed, would be seriously damaging. Parallelism is far too valuable an idea to be hidden from those trying to solve problems in the real world.

2 The Pursuit of Understanding

The real world consists of networks of autonomous interacting objects. Rich levels of abstraction are reflected in rich hierarchies of network. Data is distributed, replicated and rarely centralised. Complex global effects result from massive numbers of simple local actions.

When solving a problem in this world, a sufficiently detailed model of the relevant part must be constructed. Software design methodologies, of course, have to recognise the natural parallelism in these processes and formalise this in the early stages of design. However, they then go to considerable trouble to eliminate this parallelism in order to produce a serial model that can be expressed in a traditional serial language. This serialising phase is non-trivial and, therefore, a source of error. For all but the simplest components, the resulting executable *serial* model has no direct correspondence with the original *parallel* design, which complicates the important and expensive tasks of system verification, testing and maintenance.

On the other hand, if we can retain the parallelism inherent in the original problem all the way down to something that can be executed, we avoid the difficult serialisation effort that is standard practice today and significantly reduce costs throughout the life-cycle of the project. To enable this, we have to allow and encourage the application engineer to make full use of the concept of parallelism, which is exactly the opposite strategy to that suggested in the preceding section.

3 The Immaturity of Parallel Processing

The following statements have widespread credence in the computing community:

- parallel processing is harder than serial processing;
- parallel processing is inherently non-deterministic, so that obtaining trustworthy (i.e. repeatable) results cannot be guaranteed;
- parallel processing introduces a whole new range of problems for the designer (e.g. deadlock and livelock);
- parallel processing is very difficult to debug – a consequence of all of the above!
- parallel processing is very hard to port between parallel architectures.

These opinions run counter to the argument made in section 2. A problem with trying to resolve such disagreement is the current *immaturity* of parallel processing.

This is evidenced by the many competing paradigms and technologies. No winner has emerged around which a consensus can form to define the term.

On the hardware side, parallelism is everywhere. Up to the level of a single processing chip, there is agreement on fundamental concepts. Serial computing is a *mature* engineering discipline and hardware parallelism to provide efficient support is well developed.

For multi-processors, ideas sharply diverge into SIMD, SPMD and MIMD processing styles. They interact via shared memory (protected by semaphores, barrier synchronisation, cache-coherency hardware, ...) and/or message-passing (buffered/unbuffered, synchronous/asynchronous, point-to-point/broadcast/many-1, ...). We have a wealth of alternative technology!

Hardware vendors offer tightly-bound small clusters of powerful vector-processors, massively parallel scalar processors and DIY multi-processor networks for embedded real-time applications. Funding agencies are keen for would-be users of parallel computers to use up all that idle-time on workstations, by allowing them to be used as part of a dynamically re-shaping background compute resource.

On the software side, we have paradigms such as data-parallel and message-passing. We have functional methods for process decomposition, client-server principles, barrier synchronisation, farming and remote procedure cells. We have structured messaging systems (e.g. occam[1] and Linda[2]) and unstructured ones (e.g. PVM[3]).

At present, the choice of parallel hardware platform we make (or is made for us) severely constrains our choice of parallel software paradigm. Our views on the statements at the beginning of this section will be coloured by the parallel processing technology we have been using. My experience from working over the last 12 years with the occam/CSP model (not necessarily constrained to the transputer) indicates that the first, second and fourth statements are simply untrue and that the third one is true but easily overcome[4, 5, 6, 7]. Nobody seems to have any good experiences regarding the last statement on portability.

4 The Road to Maturity

There are three areas in which parallel processing needs to improve:

- **simplicity:** we need to make it easy to develop applications for parallel hardware;
- **efficiency:** we need to make it easy to sustain a decent percentage of the theoretical processing power of a parallel machine (more than 20% would be nice);
- **portability:** we need to make it easy to develop portable applications for parallel hardware that are independent of specific features.

Unfortunately, some of these goals pull us in different directions to others. Standard packages for message-passing (e.g. PVM and the forthcoming MPI[8]) can achieve portability, but provide only a library of functions for communicating unstructured blocks of data. They operate at too low a level to contribute to the important design decisions about how to distribute and coordinate the work. Implementations vary considerably in their efficient use of the underlying architecture – many applications that are sensitive to the start-up costs for communications cannot work with any of them.

For embedded multi-processors, standard real-time kernels offer similar portability, but again provide only a library of function calls for their application interface. The semantics of such interfaces are complex and intuitive reasoning about them cannot always be trusted.

Formalisation of the semantics of message-passing and process-scheduling libraries has not been accomplished. To make this practical, the new semantic ideas need to be integrated into the application language itself. For languages that evolved before the concepts of multi-processing, this is difficult both for technical reasons and for political reasons (it breaks "standards"). Hence, we are left with low-level run-time libraries that aim at portability at the expense of introducing an element of mystery and, for some implementations, serious inefficiency.

If the application is sufficiently important (scientifically or commercially) and its performance is damaged by the use of "portable" libraries, the engineer will invest time in learning how to exploit directly the specific architectural features of the parallel processor being targeted. Portability will be lost but performance may improve dramatically. Unfortunately, we will still be operating with parallel concepts at a low level, whose semantics are informal, imprecise and not integrated with the language used to express the "main" algorithm. We are still faced with the problem of verification, as well as long learning curves, development times and maintenance costs.

Setting *efficiency* or *portability* as the primary goal for parallel computing yields only short-term benefits. The long-term solution is to pursue the goal of *simplicity*. As ever, simplicity is the most difficult goal to achieve, sometimes requiring a radical reappraisal of basic strategies. For parallel computing, simplicity requires a clear integration of parallel concepts into all levels of system design *and* implementation. If this is well-done, efficiency and portability will be natural consequences.

5 Reconciling Parallel Software and Hardware

We reconcile the different points of view outlined in sections 1 and 2 with the following thesis. The applications engineer must be:

- **exposed** to the parallel nature of the application itself, which must be explicitly preserved throughout design and implementation;
- **shielded** from the parallel nature of the hardware to which the system happens (today) to be targeted.

To achieve the former, we must welcome the concept of *process* as a first-class engineering abstraction – in the same way as concepts such as loops, procedures and data-structures have become welcome as first-class abstractions. The notion of process must also include the idea of *synchronisation* and *communication*. Parallel processes must default to a *deterministic* semantics – i.e. the communication model must be *secure*, whether this is by message-passing or shared-memory. *Non-determinism* must be allowed, but its introduction must be explicit and only to meet specific needs in the application.

The term "first-class abstraction" implies integrating the concept as part of the syntax and semantics of the design-and-programming language. As opposed to access via run-time libraries, the semantic ideas can then be formalised and made to correspond with engineering intuition. By giving the engineer the fluency to express parallelism (at any level of design and implementation) with the same ease as (say) a conditional or a loop, we allow that expression to become more closely aligned with the application and simplify the whole task of deriving and verifying and maintaining that expression.

But the parallelism inherent in that expression will be in terms of the application and not in terms of the target hardware. The parallelism of the former may be completely different to the parallelism of the latter – for example, in the granularity of process and communication and in the network topology. Allowing and reconciling this divide is one of the *Grand Challenges* facing parallel computing.

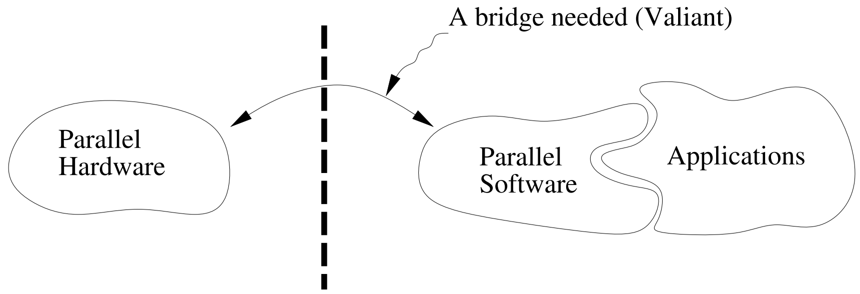

Figure 1: Reconciling parallel hardware and software

Valiant[9, 10] describes this as building a "bridge" (see Figure 1) between parallel hardware and parallel software that enables the efficient mapping of each on to the other, regardless of the underlying styles on either side. He points out that this problem has long been solved in the mature engineering discipline of serial computing (see Figure 2). The concept of the Von Neuman machine has enabled many different styles of serial hardware and software to support each other with great efficiency.

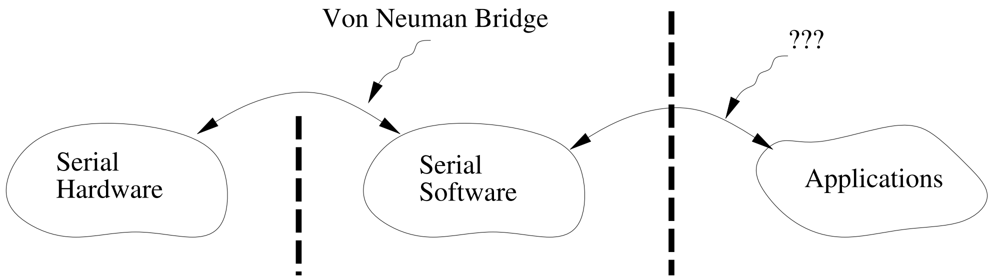

Figure 2: Reconciled serial hardware and software

The serious problem for serial computing is bridging the rather different gap between the natural parallelism in all application domains and the serial nature of the software solution. As our ambitions grow for the complexity requirements in the attempted application, the necessary bridge building becomes more and more difficult.

The serious problem for parallel computing is bridging the gap between parallel software and hardware, but we have only just started to consider this and the prospects look good. Valiant has proposed the concept of Bulk Synchronous Parallelism (BSP) as possible material for this bridge. Theoretical analysis in [9] indicates that efficient scalable mappings are possible, subject to the constraint that the parallelism in the software is *considerably* greater than that in the hardware (and where the extent of that consideration is quantified in terms of simple parameters of the parallel hardware and software). A short and light introduction to BSP may be found in [11].

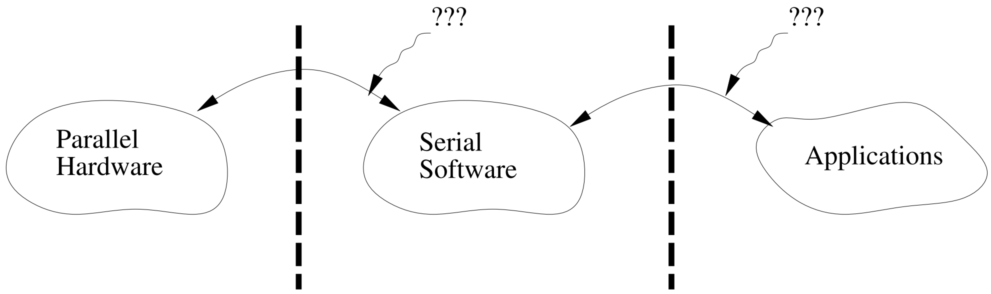

Figure 3: Worst of all worlds

Pressure from the mainstream computing community is only for the second part of the thesis presented at the start of this section (shielding from the parallel hardware). This pressure is for a very different kind of reconciliation – one between parallel hardware and serial software (see Figure 3). The grounds are short-term but massive: preservation of existing investment in the development of very large serial codes. This is coupled with anxieties concerning the statements at the start of section 3, which leads to worries about the costs of re-training existing people to the necessary skills for parallelism. However, the hidden costs of bridging the gap between applications and serial software and the unknown costs of bridging serial software to parallel hardware make this a most risky scenario. Technically, the prospects for constructing the latter bridge do not seem credible – the starting conditions completely violate Valiant's constraint for parallel slackness.

6 General Purpose Parallel Computing

It is not necessary to demand that Valiant's bridge makes an efficient connection between *all* forms of parallel software and hardware. Certain rules of good behaviour (that our current immature technologies do not obey) may be essential for both parties.

On the software side, these rules would include the deterministic behaviour of processes (unless non-determinism is explicitly introduced), a formal semantics that corresponds to our intuition and no constraints on the level at which parallelism may be applied.

On the hardware side, there needs to be a balance between computational power and communication bandwidth on the one hand, and the time-costs of process synchronisation, message-startup and context-switching on the other. Just providing very powerful processing nodes and high-bandwidth communications will *not* automatically yield high performance applications.

However, given good behaviour all round, we can begin to see how this bridge might be built.

6.1 *CSP, occam and the transputer*

The algebra of Communicating Sequential Processes (CSP)[5] provides a rich formal theory for the specification and verified implementation of parallel processes. occam provides an efficiently implementable subset of CSP, together with strict anti-aliasing and parallel usage-checking rules, that already satisfies the good-behaviour rules for parallel software. However, occam was designed primarily for embedded applications and currently lacks many features that would make it more acceptable for general purpose computing – although this is being addressed [1, 12, 13].

Professor Hoare gave the following definition[14] for a generic transputer:

> **transputer**, *n*: a computing device in which communications are primitive operations (like addition) and in which parallelism is a primitive construct (like the conditional);

This definition is clearly satisfied by the INMOS (now SGS-Thomsom) family of transputer. However, the necessary primitives for processes and communications *can* be efficiently added to most RISC architectures with an elementary software layer. Systems built on this layer directly exploit transputer concepts – i.e. to work with the transputer model, we are not constrained to work with the products of any one company.

6.2 *Physical links*

First generation (T2/T4/T8) INMOS transputers enable us to construct simple, verifiable and efficient parallel systems based on purpose-built hardware – see Figure 4.

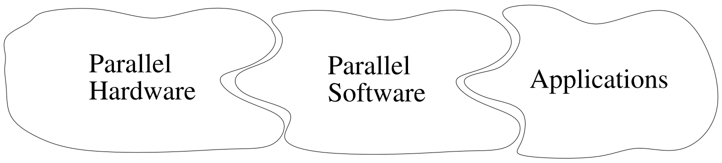

Figure 4: First generation transputer system

There is no bridge to build between the parallel software and hardware, because the hardware is tailored to supply precisely the parallel structures defined by the software. There is no bridge to build between the parallel software and the application – for reasons already described. This is an excellent state of affairs, provided we can live with the constraints.

The major constraint is the limited connectivity directly on offer from the transputer parallel hardware – although it must be pointed out that the four links provided from each INMOS transputer is four more than that provided by most other micro-processors. However, if our parallel software requires a processing node with more than four input or four output channels, then we do have to find a bridge between our software and transputer networks.

The concepts for this bridge are message multiplexing and routing. Two such bridges [15, 16] are in place, one of them[15] commercially supported within the current INMOS Toolsets for occam and C. The costs of using these software bridges ([15] more than [16]) can be high – they increase communication latency and decrease bandwidth. For some applications, this causes real difficulty.

6.3 Virtual links

Second generation (T9000) INMOS transputers provide hardware support for the bridge just described. Application messages are chopped into small (32 byte) packets and multiplexed (at this packet level) on and off a transputer through its on-chip Virtual Channel Processor (VCP). Total connectivity is provided through a fixed-topology deadlock-free routing network constructed from packet-level switches (e.g. the 32-linked C104) – see Figure 5.

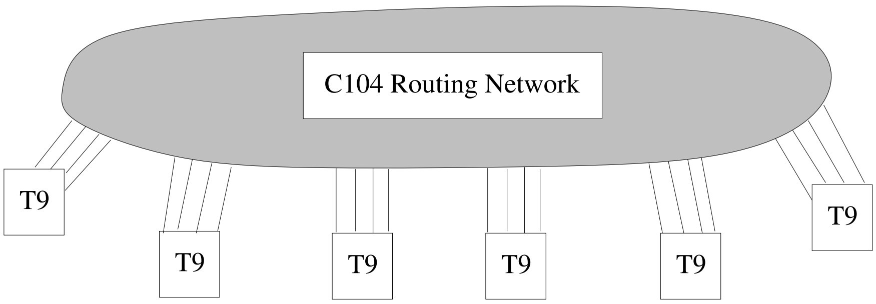

Figure 5: Totally Connected Routing Network

The number of physical transputer links is no longer relevant to the application engineer, merely "how much bandwidth do they supply?" and "what is the start-up cost for a message?". The VCP will multiplex as many channels as the application needs. In fact, it is optimised to support many channels – running only one channel per link will not obtain full bandwidth[17].

Any process on any processor can communicate with any process on any other processor by a single direct (occam) channel. Full (occam) synchronisation on each application-level message is automatic. Any processor can support any number of external channels (i.e. virtual links) – we are not limited to 4. *The software topology is independent from the target hardware topology* and can be developed entirely to suit the application. Virtual links bridge the gap between this parallel hardware and software.

Because this bridge is built into hardware (rather than software), the run-time overheads for using it are minimal. Indeed, the flexibility that results from using it are so great that we should consider doing so even when not strictly necessary.

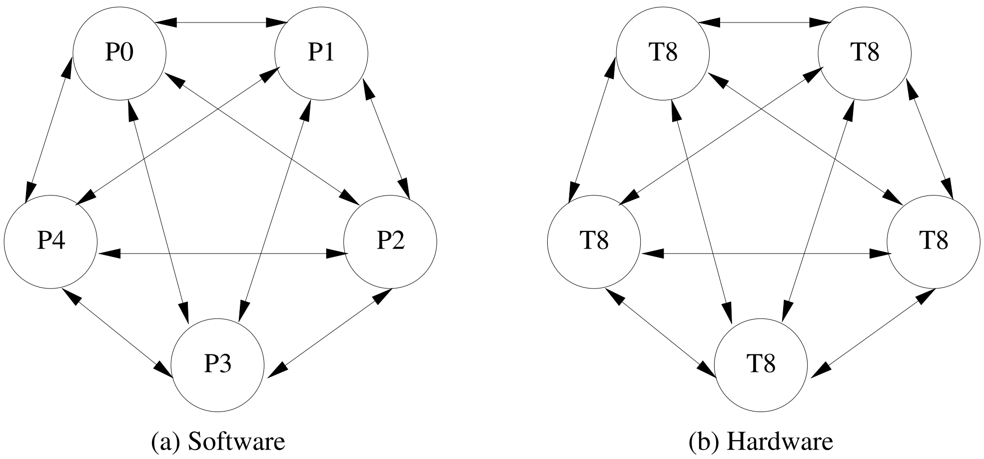

Figure 6: Some Application Software With Matching Hardware

Consider the application software design in Figure 6(a). It is a totally connected network of 5 processes that can be directly implemented with an identical network of 5 four-linked first generation transputers (Figure 6(b)). Even though this could be repeated with second generation (T9000) transputers, there are significant advantages in using the very different hardware topology shown in Figure 7:

- the system is not closed;
- if P0 and P3 (say) only want to speak to each other at some stage in the computation, they have four times the link bandwidth at their disposal. If they were connected in the application topology of Figure 6(b) they either have to make do with one link or ask the other processors to help (which would bring in other run-time overheads);
- the hardware topology of Figure 7 easily extends to support a totally connected application topology of more than 5 processes. The hardware topology of Figure 6(b) does not;
- the hardware topology of Figure 7 is not oriented to this application and can be re-used.

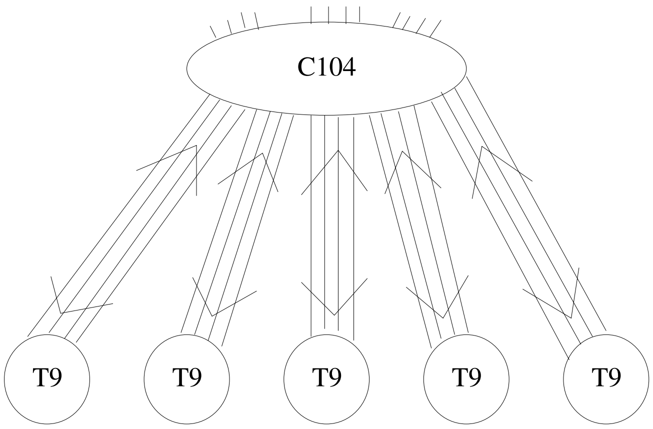

Figure 7: (T9000) Hardware Platform

6.4 Virtual links are not enough

We are still some way from building a bridge to support general purpose parallel software on parallel hardware. Even though topology is no longer a factor, we still have to allocate processes in the application network to processors in the hardware network and this allocation is static. This means that we still have to analyse and balance the computational and communication loads. This requires knowledge of both application software and hardware – a change in one will require a new allocation to be worked out. If the loads vary dynamically, the fixed allocation will not always be optimal.

To complete the bridge between parallel software and hardware, we must eliminate the necessity of finding – and the inefficiencies of running – a fixed process/processor mapping.

7 Virtual Shared Memory

Physically shared memory between multi-processors over a single bus does not scale, because the bandwidth on the bus is finite. Private memory multi-processors *can* scale, provided that we increase the communication fabric that links them together at the same time as (and slightly faster than) we increase the number of processors.

Second generation transputers give us that scaling potential (because the network fabric and processing nodes are constructed from different chips and can, therefore, be resourced in different ratios). However, they do not provide a shared memory model.

The reason we are interested in shared-memory multi-processing is not because of any supposed benefit that model may offer for application software, but because of the prospect it offers for eliminating the need for a fixed process/processor mapping and, hence, completing the bridge between parallel software and hardware.

7.1 Cache Coherency

A serious problem for any shared-memory multi-processor (whether real or virtual) is that of *cache coherency*. Caching has been successfully developed for single processor computing to tolerate the huge discrepancy between the latencies for register-register transfer and register-memory transfer. Often two levels of cache are introduced.

Multi-processors offering shared-memory must provide a consistent view of that memory to all processes, regardless of where they happen to be executing. If two processes running on different processors cache in locally the same segment of shared memory and one of them makes a change to its local cache, the two processes are suddenly looking at inconsistent versions of the supposedly-same thing. The caches have become *incoherent* and something needs to be done.

If we panic at this point, everything gets very expensive! First, we will have to set up some (hardware) mechanism to let each processor find out if any of its local cache-lines become invalid because of action in some other processor. We may also need to let each processor know if its actions on its local cache are invalidating cache-lines elsewhere. The capital hardware costs and the run-time costs for this operation are significant. Furthermore, once incoherency has been detected, cache-lines will have to be flushed and re-filled. These run-time costs for “badly behaved” parallel applications may completely eliminate the caching benefits to a processor (that it normally obtains when operating as a stand-alone serial machine). Since the performance of our fastest micro-processors depends critically on the success of its caching regime, we can have a real problem trying to use them for multi-processing.

What needs to be done is not to leave it to run-time to sort out the problems of cache incoherency. For well-behaved software, most problems can be eliminated before the code hits the machine. If we insist that parallel software operates in (say) *Concurrent-Read-Exclusive-Write* (CREW) mode for PRAM architectures[18], we know some highly relevant information. For example, occam processes are guaranteed (by the language rules and enforced by the compiler) to operate CREW and so:

- any data modified by one process cannot be viewed by any other process running in parallel;

- any data being viewed by one process may also be being viewed by other processes, but cannot be altered by any of them.

Hence, whilst a process has cached in some data to a cache-line in some processor, *absolutely no action need be taken to ensure the coherency of that cache-line*. Even if that data area is cached on some other processor and the process responsible is altering some part of that cache-line, it must be a part of the cache-line that the first process is not viewing – so there is no problem! There are only two points that must be addressed that are different to the caching arrangements for a uni-processor:

- when a process terminates or deschedules, any cache-lines in use by that process must be flushed. These new flushing points can be coded explicitly by the compiler. [Note: the reason for flushing at a descheduling point is to enable the process to be picked up later by a different processor – see below].

- when a cache-line is flushed, only those parts that have been changed must be written to memory. This implies maintaining a *changed-bit* for each unit of store-granularity (typically a byte or word) in the cache-line, rather than just one *changed-bit* for the whole line. Whilst this is an overhead, it is a very modest one in terms of hardware and zero in run-time.

Thus, no run-time monitoring for cache consistency is needed and only a few extra (pre-determined) flushes are needed compared to the requirements for a uni-processor computer. The nodes of a (virtual) shared-memory multi-processor receive almost all the benefits from their local cache and can, therefore, run at almost full speed – the slight loss being greatly compensated by the benefits of dynamic process allocation.

7.2 Dynamic processes

Construct a parallel machine from a collection of processing nodes, memory nodes, switching nodes and links. Nodes (of any variety) are only connected via links. In particular, there is no intimate association between a processing node and any memory node. The memory nodes present a single memory space – the same one to each processing node. There must be (roughly) equal-cost time access from any processing node to any memory node. This may require the mapping of logical memory addresses (in units of cache-line) on to physical memory nodes via a "good" (hardware implemented) hash function. The processing nodes will include fast local caches and may include multiple processing pipelines. Randomised routing, message broadcasting and message combining may be employed in the switching fabric to assist even loading on that fabric.

Such a machine has removed any benefit that might have been obtained by exploiting-application locality of data. Logically contiguous memory blocks will be scattered across physically different memory nodes. All processors will be equally efficient at executing a process, regardless of the data area on which the process is working. Therefore, not only do we not care which processor runs which process, but we also do not care in which memory nodes the workspace for a process is allocated.

The *run-queue* of processes (that are ready to continue execution) lives in memory. There is only one memory – i.e. all processors can see this queue. Use the collection of processors as a *farm* – whenever a processor runs out of work, it removes a process from the head of the run-queue. If a processor does something that enables a blocked process to continue, it adds it to the back of the run-queue. Removing and adding processes to this queue will have to be securely synchronised, as all processors may be trying to do this simultaneously.

There are a number of interesting benefits arising from this hardware scheme:

- a process, when it gets (near) to the front of the run-queue, will be picked up and executed by *any* processor needing more work. A process is not statically pre-allocated to a processor. It continually "migrates" to a lowly-loaded processor;
- message-passing has equal-cost to assignment – i.e. there is no overhead to be paid for message-passing compared to that paid for assignment. This is because message-passing is just a distributed assignment and, since memory is now (hash) distributed, the switching fabric will need to be employed equally for both types of operation;
- large data-transfers (caused either by message-passing or assignment) can be managed entirely by the switching fabric, with no processor involvement;
- large data-transfers can proceed in parallel (because a single logical block of data will be spread over many physical memory nodes and will be aimed at many physical memory nodes);
- a channel has no pre-determined allocation to some fixed route through the network. The start and end points will depend on the source and target data-structures in the communication. This will vary during run-time, further spreading the load on the routing fabric.

The above gives us automatic load-balancing of parallel software on parallel hardware – there is no need to analyse the actual computation and communication structures in the software and no need to balance against the capabilities of the hardware. We only need to ensure that the parallel software is well-behaved (see section 6) and that there is sufficient software parallelism to keep the run-queue (a bit) longer than the number of hardware processors. So long as this applies, the more processors we add, the faster the program will run – we will get the scaling efficiencies of the *farming* paradigm (which can exceed 100%) regardless of the parallel paradigms actually used in the application.

Processing nodes may be multi-processor internally. In that case, each node will be managing several processes at a time – where "several" will again exceed the number of local processors. This may necessitate a local run-queue held in its local "cache". The advantage is that when a local cache miss occurs, the local process responsible can be suspended and another local process switched in very quickly. When the missed cache is ready, the suspended process is added to the local run-queue. Processes migrate between such local queues and the main run-queue either on an age-basis or at random.

8 Conclusions

We look forward to parallelism becoming another option in the general-purpose (personal) computer/workstation marketplace. Not only will we be offered a choice of clock-speed for our processor, but we will be able to decide how many (1, 10 or 100?) we can afford. The system will always be upgradable through the addition of processors, memory and switches. Any application, provided it is big enough and implemented with sufficient parallelism, will scale (possibly super-linearly) in performance in line with the money we spend. The same object code will be used, regardless of the number of processors executing it.

This vision is different to that currently offered by multi-processor workstations. These make a *fixed* allocation of large-granularity process to processor, using a strategy dependent on the processor loadings at the time of the allocation. This is unlikely to yield continuously even loading on the machine's resources and, hence, will lead to processor idle-time when there is still work to be done. An extreme – but typical – case is when there is only one large (serial) application to run and we can only exploit one of the processors!

Our vision is also different to that provided by the shared-memory SPMD paradigm. This similarly requires a *fixed* allocation of process to processor. Only one process algorithm is allowed, so it is the data-structures on which the algorithm operates that are distributed to the processors. The logic for this single process is serial and the data-parallelism has to match the hardware-parallelism. We still have to balance the software demands on the parallel hardware and accept inefficiency if this balance varies during run-time (or just cannot be found). We also have to find and cross the bridge between the natural parallelism in the original application and the hardware-oriented (data-parallel, logic-serial) software.

On the other hand, we have described a complete separation between parallel hardware structures and the software that runs on it. This permits software structures to be created solely in terms of the application, subject only to the constraints that they be highly parallel and well-behaved (e.g. CREW and subject to the laws of CSP). We claim that this latter is no constraint and, in fact, is necessary for the simple and verifiable modelling of complex systems.

The bridge between the application-oriented parallelism in the software and the parallelism in the hardware depends upon excess software parallelism, "good behaviour" from that software, no fixed process-processor allocation, no need for run-time cache-coherency monitoring, no overhead for message-passing (compared with assignment), parallel data-transfer and very fast context-switching.

On the last point, the INMOS transputer demonstrated sub-microsecond context-switching ten years ago (1985) using just 20 MHz parts! This was in the context of also supporting interrupt handlers (triggered by link traffic and external devices), processes waiting on time-outs and (2-level) priority pre-emptive scheduling. This performance has still not been approached by more "state-of-the-art" micro-processors. The transputer has much to teach us about the organisation of process state so as to achieve (ideally) single-cycle process management.

The final consequence of this is that our future "ParaPC" will run devoid of a conventional operating system. Applications will be fine-grain parallel and will schedule themselves *fairly* across all processor nodes, with no software control. "Operating system" functions will be unbundled into optional process libraries (e.g. for user interfaces, file servers, network servers, ...) to be included at the discretion of each application.

We have outlined ways to reconcile parallel software and parallel hardware and, hence, open up parallel processing to support efficiently general-purpose computing. Technology has reached the point where the hardware end of the bridge can start to be built. The software end lies in terrain to which our existing vast baggage of code cannot easily be carried. We will have to wait for the need for that bridge to become greater than the need to preserve old ways of working. Don't hold your breath.

9 Acknowledgements

Many of these ideas have been voiced in public talks given by David May, Roger Shepherd, Peter Thompson (of SGS-Thomson Microelectronics) and others over the past few years. I am not aware of any formal publication.

I am grateful to all my colleagues in University and Industry for numerous private conversations that have helped me to understand these concepts. However, any resemblance to any project that may be in progress in any commercial organisation would be purely coincidental.

I am especially grateful to David Beckett (University of Kent) for getting this paper to press, despite a broken arm, and for critical review.

References

[1] Geoff Barrett. occam 3 *reference manual (draft)*. INMOS Ltd, Bristol, BS12 4SQ, England, March 1992. See URL http://www.hensa.ac.uk/parallel/occam/documentation/manual3.ps.Z.

[2] N. Carriero and D. Gelernter. How to write parallel programs: A first course. Technical report, MIT Press, Cambridge, MA, 1990.

[3] V.S. Sunderam. PVM: A framework for parallel distributed computing. *Concurrency: Practice & Experience*, 2(4):315–339, December 1990.

[4] P.H. Welch. Emulating digital logic using transputer networks. In *Proceedings of the PARLE International Conference*, volume 258 of *Lecture Notes in Computer Science*, pages 357–373, Eindhoven, June 1987. Springer-Verlag.

[5] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[6] P.H. Welch, G.R.R. Justo, and C. Willcock. High-level paradigms for deadlock-free high-performance systems. In Grebe et al., editors, *Transputer Applications and Systems '93*, pages 981–1004, Amsterdam, 1993. IOS Press.

[7] Jeremy Martin, Ian East, and Sabah Jassim. Design rules for deadlock freedom. *Transputer Communications*, 2(3):121–133, September 1994.

[8] MPI Forum. MPI: A message-passing interface standard. Technical report, Message Passing Interface Forum, May 1994. See URL http://www.mcs.anl.gov/mpi/mpi-report/mpi-report.html.

[9] L.G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, August 1990.

[10] L.G. Valiant. Bulk-synchronous parallel computers. In M. Reeve and S.E. Zenith, editors, *Parallel Processing and Artificial Intellligence*, pages 15–22. Wiley, 1989.

[11] Michael G. Norman. Bulk synchronous parallelism. *WoTUG Newsletter*, 17:32–35, July 1992.

[12] Jon Kerridge. Using occam3 to build large parallel systems: Part 1, occam3 features. *Transputer Communications*, 1(1):47–63, August 1993.

[13] Jon Kerridge. Using occam3 to build large parallel systems: Part 2, a scalable file system. *Transputer Communications*, 2(1):1–24, March 1994.

[14] R. Grebe et al. Editorial. *Transputer Communications*, 1(1):1–2, August 1993.

[15] Mark Debbage, Mark B. Hill, Denis A. Nicole, and Andrew Sturges. The virtual channel router. *Transputer Communications*, 1(1):3–18, August 1993.

[16] J. Sunter, A.W.P. Bakkers, and Klaas C.J. Wijbrans. Virtual channel generator – VCG. In R. Grebe et al., editors, *Transputer Applications and Systems '93*, volume 1, pages 336–348, Amsterdam, September 1993. IOS Press.

[17] M.D. May, P.W. Thompson, and P.H. Welch. *Networks, Routers and Transputers*. IOS Press, Amsterdam, 1993.

[18] U. Vishkin and A. Wigderson. Trade-offs between depth and width in parallel computation. In *24th Annual Symposium on Foundations of Computer Science*, pages 146–153, 1983.