Eurographics Symposium on Parallel Graphics and Visualization (2004)
Dirk Bartz, Bruno Raffin and Han-Wei Shen (Editors)

Through the Concurrency Gateway: a Challenge from the Near Future of Graphics Hardware

P. H. Welch

University of Kent Computing Laboratory, UK.

Abstract

*The computer graphics industry, and in particular those involved with films, games and virtual reality, continue to demand more and more realistic computer generated images. The complexity of the scenes being modelled and the high fidelity required of the images means that rendering is simply not possible in a reasonable time (let alone real-time) on a single computer[BrW03]. Interactive ray tracing exists today[WSB*01], but real-time global illumination remains a major challenge. Fortunately, "computer graphics cards are developing at Moore's law cubed" [David Kirk, Chief Scientist, nVIDIA]. Such performance increases are directly due to the inherent parallel nature of modern graphics cards. If this trend continues, they will be 100 times faster in a mere 3.5 years time, 1000 times faster in 5 years and they will be massively parallel. Unfortunately, past experiences in designing systems that can exploit parallel processors in anything beyond embarrassingly trivial ways are not encouraging. For real-time interaction with high fidelity images, the parallel processing requirements will not be embarrassingly trivial! Regular and irregular patterns of synchronisation and communication will have to be managed over networks of fine-grained (for accuracy) model components whose scale, topology and physical distribution are dynamically evolving. This paper reviews weaknesses in our standard approaches to the design and implementation of concurrent systems and describes ways forward that are mature and practical – both for the programmer to program and the hardware to execute. They are built on decades of research into process algebrae (CSP and the π-calculus), but are able to preserve and exploit traditional skills and capabilities of serial software engineering and von Neumann architecture (components of which will still form the processor base of parallel systems for at least the next decade). The changes are, therefore, evolutionary rather than revolutionary – but are nevertheless essential both in the field of graphics and for the wider Grand Challenges of computer science.*

Categories and Subject Descriptors (according to ACM CCS): I.3.1 [Computer Graphics]: Parallel Processing

1. Introduction

1.1. Massive parallelism – beyond the von Neumann

Modern silicon fabrication has delivered astonishing increases in the density of logic that can be accommodated on any given chip. Soon, the only credible way of exploiting these riches is to make that logic *parallel* – we simply do not know how to design a *serial* processor complex enough to use all the real estate. Filling the silicon with replicated processor *'cores'* is the obvious solution, providing we lay in sufficient and efficient mechanisms to let them synchronise, communicate and share resources. That still leaves plenty of arguments regarding the style and granularity of this parallelism – from the very fine-grained (e.g. FPGAs), through lock-step data-parallel (SIMD, vector) to the relatively chunky (on-chip SMP and MIMD). Graphics engines will follow one or more of these routes and graphics engineers will be amongst the first in having to deal with it.

Hiding parallelism in the hardware from those building applications is not going to be easy. Programming languages and tools have been highly successful at shielding their users from the intricacies of serial hardware (such as the number and type of registers, levels of cache hierarchy, out-of-order code execution). This reflects the success of the von Neumann paradigm in providing a *'bridge'* between serial software and serial hardware. As Valiant proposed[Val90], we need to build such a bridge between parallel software and parallel hardware. Note, however, that this does not imply finding magical mappings from serial software on to paral-

lel hardware (and which would mean we would not have to change our ways). These will probably never be found and that is a very good thing – *parallelism is far too valuable an idea to be hidden from those trying to solve real world problems*.

This paper outlines a framework for dealing with a variety of parallel hardware architectures likely to be on offer over the next few years. The ideas embrace concurrency as a friend ... as ways to structure both the design and implementation of systems that are natural, open to analysis, intuitive and efficient. They preserve all the skills we have developed for working with serial (von Neumann) software, allowing them to used independently from parallel aspects of the system. They reflect similar alliances in hardware, where the developments in serial processor design are simply too good to discard.

1.2. Concurrency is natural

The real world exhibits concurrency at all levels of scale - from atomic, through human, to astronomic. This concurrency is endemic. Central points of control do not remain stable for long. Our experiences with building and interacting with complex systems hint at something stronger - namely that central points of control actively work against the logic and efficiency of whatever it is that is we are trying to control/model/understand and that, in the long term, we must give up on it. The case made here is that it is necessary to give up on it now, that it is possible to do so and that it will be profitable to do so. Indeed, our ability to harness next generation hardware depends on it. But a mindset needs to be changed and that requires a lot of pushing.

1.3. Current mechanisms are unnatural

In present day computer engineering, concurrency is not considered a fundamental concept – to be used everyday with the same fluency as we might use, say, *classes* or *while-loops*. It is taught, almost universally, only as an advanced topic and only to be used when there are no other ways to obtain specific performance targets. Examples include the reduction of response times to external interrupts/commands (whilst long running background computations are continuing), or the speed-up of completion times for large-scale scientific or engineering calculations (through the use of multiprocessors). Both of these, of course, are relevant for interactive high-fidelity graphics.

Standard concurrency technologies are based on multiple threads of execution plus various kinds of locks to control the sharing of data between them. Get the locking wrong and systems will mysteriously corrupt themselves or deadlock. Received wisdom from decades of practice is that concurrency is very hard, and we are advised to steer well clear if at all possible [MW00].

In addition to these logical problems, there are also performance problems. Standard thread management imposes significant overheads in the form of additional memory demands (to maintain thread state) and run time (to allocate and garbage-collect thread state, to switch processor context between states, to recover from cache misses resulting from switched contexts, and to execute the protocols necessary for the correct and safe operation of locks). Even when using only 'lightweight' threads, applications need to limit their implementations to only a few hundred threads per processor – beyond which performance catastrophically collapses (usually as a result of memory thrashing).

1.4. Our ambitions leave us no choice

Modern computing already faces a dilemma: it is driven by ever-increasing demands for system functionality, performance, responsiveness, inter-operability, dynamics, safety, and security. Yet our standard concurrency models and tools, which ought to be fundamental in addressing these demands, throw up serious new problems that act against them. As a result, concurrency is used on a relatively small scale, where its analysis is (just) manageable and the performance benefits outweigh the overheads.

But the problems – and our ambitions, even now – are much bigger than this. For example, the visualisation and control of air traffic over the UK requires the management of far greater concurrency than standard practice will directly and safely and simply allow. Common web services need to be able to conduct business with tens of thousands of clients simultaneously. Modelling even the simplest biological organisms quickly takes us into consideration of millions of concurrently active, autonomous, and interacting, agents. Limited by such constraints, we have to compromise on the degree of concurrency in our application design and implementation. Those compromises add significant complexity that, combined with the semantic instability of the concurrency mechanisms we do practice, lead to mistakes and the poor quality, late delivery and over-budget systems that are accepted as normal – for now – by our industry and its customers.

This submission suggests some ways for leaving these constraints behind.

2. A thesis and a hypothesis

All computer systems have to model the real world, at some appropriate level of abstraction, if they are to receive information (data, signals, etc.) and feedback useful information (reports, control, etc.). To make that modelling easier, we should expect concurrency to play a fundamental role in the design and implementation of systems, reflecting the reality of the environment in which they are embedded. This does not currently seem to be the case.

Our thesis is that computer science has taken at least one wrong turn. Concurrency should be a natural way to design any system above a minimal level of complexity. It should simplify and hasten the construction, commissioning, and maintenance of systems; it should not introduce the hazards that are evident in modern practice; it should be employed as a matter of routine. Natural mechanisms should map into simple engineering principles with low cost and high benefit. Our hypothesis is that this is possible.

2.1. Combining CSP and the π-calculus

We propose a computational framework, based on established ideas of process algebra, to test the truth of the above hypothesis. It will be accessible from current computing environments (platforms, operating systems, languages) but will provide a foundation for novel ones in the future. It will integrate the best ideas from Hoare's *Communicating Sequential Processes* (CSP)[Hoa78, Hoa85, Ros97] and Milner's π-calculus[Mil99], though this will require additional work on the theory.

CSP has a compositional and denotational semantics, which means that it allows modular and incremental development (refinement) even for concurrent components. In turn, this means that we get no surprises when we run processes in parallel (since their points of interaction have to be explicitly handled by all parties to these interactions). This is simply not the case for standard threads-and-locks concurrency, which have no formal denotational semantics and by which we get surprised all the time.

However, we need some extensions to describe certain new dynamics – and this is where we turn to the π-calculus. Specifically, we want to allow networks of processes to evolve, to change their topologies, to cope with growth and decay without losing semantic or structural integrity. We want to address the mobility of processes, channels and data and understand the relationships between these ideas. We want to retain the ability to reason about such systems, preserving the concept of refinement.

2.2. Testing the hypothesis

The framework has to provide highly efficient practical realisations of this extended model. Its success in targeting future parallel graphics hardware will be one long term test of the above hypothesis. Shorter term tests will be the development of demonstrators on current platforms with the following characteristics:

- they will be as complex as needed – and no more (e.g. through the concurrency in the design being directly delivered by the concurrency in the implementation);
- they will be scalable both in performance and function; *[Note: by functional scalability, we mean that the cost of incremental enhancement depends only on the scale of that enhancement – not upon the scale of the system being enhanced. The latter is the present state-of-the-art and is a major reason behind system delay and eventual failure.]*
- they will be amenable to formal specification and verification;
- notwithstanding the above, the concurrency models (and mechanisms) in their design (and implementation) will be practical for everyday use by non-specialists – concurrency becomes a fundamental element in the toolkit of every professional computer engineer;
- they will make maximum use of the underlying computation platform (through significantly reduced overheads for the management of concurrency – including the response times to interrupts).

3. Current state of the framework

Over the past ten years, our group[Wel04c] at Kent has been devoted to laying the foundations for such a framework. We have developed – and released as open source – concurrency packages for the Java (JCSP), C (CCSP), C++ (C++CSP) and J# (J#CSP) programming languages [Wel04a, WAF02, WV02, Wel00, BrW03, Qui04]. Despite their names, they all provide the mobile dynamics fed in from the π-calculus (although it is easy to mis-program them, since their base languages do not have a clue as to what is happening). We have also advanced the original CSP programming language, occam, to do the same – but with some major safety and performance benefits (because the base language does know what is happening). An overview of the current state and potential of this language (christened, for the moment, occam-π) is given below. More detailed overviews of this work (on JCSP and occam-π) can be found on-line at [Wel04b].

occam-π is a sufficiently small language to allow experimental modification and extension, whilst being built on a language of proven industrial strength. It integrates the best features of CSP and the π-calculus, focussing them into a form whose semantics is intuitive and amenable to everyday engineering by people who are not specialised mathematicians – the mathematics being built into the language design, its compiler, run-time system and tools so that users benefit automatically from that foundation. The new dynamics broadens its area of direct application to a wide field of industrial, commercial and scientific practice.

occam-π runs on modern computing platforms and has much of the flexibility of Java and C, whilst at the same time retaining all the safety guarantees of classical occam (e.g. against aliasing and parallel usage errors) and the lightness of its concurrency mechanisms. It supports the dynamic allocation of processes, data and channels, their movement across channels and their automatic de-allocation (without the need for garbage collection, which otherwise invalidates real-time guarantees) [BW04, Bar04, BaW03, BW01, SBW03, BJV03]. We have

extended the range of static safety checks so that aliasing errors and race hazards are not possible in occam-π systems, despite the new dynamics. This means that subtle side-effects between component processes cannot exist, which impacts (positively) on the general scalability and dependability of systems. The mobility and dynamic construction of processes, channels and data opens up a wealth of new design options that will let us follow nature more closely – with network structures evolving at run-time. Apart from the logical benefits derived from such directness and flexibility, there will be numerous gains for application efficiency.

Performance overheads for all occam-π concurrency mechanisms are mostly unit time, with the order of between 50 and 150 nanoseconds on modestly powered PCs (1GHz). Memory overheads are also very light: no more than 8 words per process. This means that dynamic systems evolving hundreds of thousands of (non-trivial) processes are already practical on single processors. Those processes can be implementing complex behaviour with time and space overheads for managing the concurrency minimal (less than 10%). Further, occam-π networks can naturally span many machines – the concurrency model does not change between internal and external concurrency. Application networks up to millions of non-trivial processes then become viable (e.g. on modest clusters of laptops). The speedup in Moore's Law predicted over the next few years for graphics accelerators means that dynamic evolving networks of hundreds of millions of processes will become possible. This enables novel approaches to model building and visualisation that promises far greater accuracy and realism than serial techniques will ever allow.

A formal denotational semantics for occam-π mobile processes, based on Hoare and Jifeng's *Unified Theory of Programming*[HJ99], has been drafted by Jim Woodcock (also at Kent) and one of his students (Xingbei Tang). However, these mobiles have not yet been implemented by the occam-π compiler and kernel, although we believe this will be straightforward. This contrasts to the status of mobile channel-ends, which are fully supported by the current system but for which a denotational semantics is still being researched.

4. One model application

With colleagues at Royal Holloway and at York, we are interested in questions about dependability and evolution for novel embedded networks that may become viable during the next 10-20 years: Nanite Assemblers. These are active devices that manipulate their world (e.g. a human body) at the nanoscopic level, but which cause macroscopic effects (e.g. through cooperating to assemble artefacts 'cell' by 'cell'). In order to be effective, vast numbers of such nanites are needed, and these numbers may grow exponentially as they assemble copies of themselves. We need the capabilities to design, model, program and control complex and dynamic networks of these machines, and to give credible assurance that they behave properly. As with swarm intelligence and ant colony algorithms, the interesting behaviour of a host of nanites comes from emergent properties. Interactive visualisation will be crucial to allow flexible experimentation.

Hierarchical networks of communicating processes are particularly suitable for these problems. But the languages used to support modelling and simulation must be simple, formal, and dynamic, and have a high-performance implementation. The models of such complex systems must be as simple as possible. The models must be amenable to manipulation and formal reasoning. The topologies of these networks will evolve dramatically, as they support growth and decay that comes from nanites moving, splitting, and combining. Individual nanites must not only be mobile, they must also be aware of their own location and the proximity of their neighbours. Finally, simulations will require very large numbers of processes, so their implementation had better have very low overheads.

Process mobility and neighbourhood awareness (so they can find each other!) requires some new thinking. We are exploring the construction of a *matrix* of processes defining the topology of the space over which mobile *agent* processes roam. The matrix nodes are (mostly passive) servers, in touch with neighbouring nodes and on which arriving agents register. An agent attaches to one matrix node at a time, through which it can sense the presence of other agents and, hence, connect and interact as it chooses (using agent-specific protocols to avoid deadlock). Matrix-agent protocols will be generic. Agents may enrol and resign from local (or global) barrier synchronisations to maintain a sense of time – as well as move and reproduce according to their own rules. Matrix nodes may also have their own agenda, allowing them to be pro-active in reshaping the space they define (e.g. through the creation of *worm-holes*) for more exotic environments.

A good candidate for modelling and programming such systems is occam-π it is robust and lightweight, and has sound theoretical support. We know how to construct systems to the order of millions of processes on modest processor resources, exhibiting rich behaviours in useful run-times. This is enough to make a start on our journey.

5. Conclusions

A *gateway event* [Gel94], described by Stepney et al. [UKC03], *"produces a profound and fundamental change to the system: once through the gateway, life is never the same again"*. Software engineers has been toying with concurrency for more than 35 years, but it has always remained the preserve of the few – a difficult and risky technology of last resort. Modern hardware architectures, with graphics engines at the forefront, are forcing concurrency into the mainstream – for routine consideration by the many. Which

places us in front of a significant gateway event in computation: that of doing business with concurrency as a *friend* and not as a *foe*.

Success requires a serious change in mindset, not all the details of which are yet clear. One that is, though, is to welcome the concept of *process* as a a first-class engineering abstraction – in the same way as concepts such as loops, procedures and data-structures have been welcomed as first-class abstractions. The notion of process must also formalise ideas of *synchronisation*, *communication* and *sharing*. It may help if processes default to a *deterministic* semantics – i.e. the communication model should be secure, whether by message-passing or shared-memory. *Non-determinism* must be allowed, but its introduction should be explicit and only to meet specific needs in the application.

Process management must be very fast and dynamic, allowing networks to be constructed, reshaped and taken down on-the-fly. De-centralisation of control will be essential, with global behaviour *emerging* from myriads of local behaviours. For full autonomy, processes themselves will need to become mobile – aware of their neighbourhood and other processes that may be only temporarily resident – and able to connect, reproduce, disconnect and relocate based on local negotiations and decision making.

We have outlined one approach to a framework for meeting these goals – doubtless there are others. We are still some way short of a full *reconciliation* between the necessarily differing aspects of parallel hardware and software [Wel95]. That will be needed to cope with rapidly changing capabilities in hardware technology that will be inherently parallel. Applications engineers will need to 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.

We are moving towards this ideal. Meanwhile, there is much to be done as we await the next generation of hardware accelerators that will force our hand. It is time to move through the concurrency gateway.

References

[BJV03] F. R. M. Barnes, C. L. Jacobsen, and B. Vinter. RMoX: a Raw Metal occam Experiment. *Communicating Process Architectures 2003*. *Concurrent Systems Engineering Series* **61**:269–288. IOS Press September 2003.

[BW01] F. R. M. Barnes and P. H. Welch. Mobile Data, Dynamic Allocation, and Zero Aliasing: an occam Experiment. *Communicating Process Architectures 2001*. *Concurrent Systems Engineering Series* **59**:243–264. IOS Press September 2001.

[BW04] F. R. M. Barnes and P. H. Welch. KRoC Home Page. `www.cs.kent.ac.uk/projects/ofa/kroc/`. 2004.

[BaW03] F. R. M. Barnes and P. H. Welch. Prioritised Dynamic Communicating and Mobile Processes. *IEE Proceedings Software* **150**(2) April 2003.

[Bar04] F. R. M. Barnes. RMoX: An occam Operating-System. `rmox.net/prelude/` 2004.

[BrW03] N. C. C. Brown and P. H. Welch. An Introduction to the C++CSP Library. *Communicating Process Architectures 2003*. *Concurrent Systems Engineering Series* **61**:139–156. IOS Press September 2003.

[BrW03] A. G. Chalmers, E. Reinhard, T. Davis. *Practical Parallel Rendering*. AKPeters, ISBN 156881-179-9, 2002.

[Gel94] M. Gell-Mann. *The Quark and the Jaguar*. Abacus, 1994.

[HJ99] C. A. R. Hoare and H. Jifeng. *Unifying Theories of Programming*. Prentice Hall 1999.

[Hoa78] C. A. R. Hoare. Communicating Sequential Processes. *Communications of the ACM* **21**(8):666–677 August 1978.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall 1985.

[Mil99] R. Milner. *Communicating and Mobile Systems: the π-Calculus*. Cambridge University Press 1999.

[MW00] H. Muller and K. Walrath. Threads and Swing *(final section: Why did we implement Swing this way?)* `java.sun.com/products/jfc/tsc/articles/threads/threads1.html` 2000.

[Qui04] Quickstone Technology Ltd. xCSP Home Page. `www.quickstone.com/xCSP/`. 2004.

[Ros97] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall 1997.

[SBW03] M. Schweigler, F. R. M Barnes, and P. H. Welch. Flexible, Transparent, and Dynamic occam Networking with `KRoC.net`. *Communicating Process Architectures 2003*. *Concurrent Systems Engineering Series* **61**:199–224. IOS Press September 2003.

[UKC03] UK Computing Research Committee. Grand Challenges for Computing Research Draft Proposals. Proposal 7: *Journeys in Non-Classical Computation*. `umbriel.dcs.gla.ac.uk/NeSC/general/esi/events/Grand_Challenges/proposals/` 2003.

[Val90] L. G. Valiant. A Bridging Model for Parallel Computing. *Communications of the ACM* **33**(8):103–111 August 1990.

[WAF02] P. H. Welch, J. R. Aldous, and J. Foster. CSP Networking for Java (`JCSP.net`). *Computational Science—ICCS 2002*:695–708. Springer-Verlag April 2002.

[WSB*01] I. Wald, P. Slusallek, C. Benthin, and M. Wagner. Interactive rendering with coherent ray tracing. *Computer Graphics Forum*, 20(3):153-164, 2001.

[WV02] P. H. Welch and B. Vinter. Cluster Computing and JCSP Networking. *Communicating Process Architectures 2002*. *Concurrent Systems Engineering Series* **60**:203–222. IOS Press September 2002.

[Wel00] P. H. Welch. Process Oriented Design for Java: Concurrency for All. *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'2000)* **1**:51–57. CSREA Press June 2000.

[Wel04a] P. H. Welch. CSP for Java (JCSP) Home Page. `www.cs.kent.ac.uk/projects/ofa/jcsp/` 2004.

[Wel04b] P. H. Welch. IFIP WG2.4 (Peter Welch's page). `www.cs.kent.ac.uk/projects/ofa/ifip/` 2004.

[Wel04c] P. H. Welch. Concurrency Research Group. `www.cs.kent.ac.uk/research/groups/-crg/` 2004.

[Wel95] P. H. Welch. Parallel Hardware and Parallel Software: a Reconciliation. *Proceedings of ZEUS'95 and NTUG'95*, pp. 287–301, Linkoping, Sweden. IOS Press, ISBN 90-5199-22-7, May 1995.