Dynamic generation of parallel computations

James Hanlon, Simon J. Hollis

Department of Computer Science, University of Bristol, UK
{hanlon, simon}@cs.bris.ac.uk

Abstract

It is now accepted that parallelism will be the primary means of increasing performance of computer systems, but the preoccupation over the last 30 years with frequency scaling of single-core systems has meant that relatively little work has taken place with regards to how we should architect and program parallel computers. Currently, there is little consensus in the approach taken with new designs, which has resulted in a growing gap between the design of languages and architectures, and the algorithms written. This paper argues that to be able to effectively use parallel systems, simple concurrency mechanisms must be provided as features of a programming language, and that these must be supported by primitive operations in the underlying architecture. In particular, the concepts of *process migration* and *parallel recursion* allow the expression of simple but powerful concurrent programs.

1 Introduction

Concurrency in computer systems is the concept of multiple *processes* executing simultaneously and interacting with each other. It has been an area of study active since the mid 1960s and today we are familiar with it as a design approach for multi-processor systems, but it was originally conceived as a key abstraction in the design of real-time systems, in particular operating systems. Edsger Dijkstra's 1965 paper *Cooperating sequential processes* [7] laid the first foundations for abstract concurrent programming, followed closely by contributions in the theory and implementation of concurrent programming principles, from Tony Hoare and Per Brinch Hansen; for examples see [6, 10, 16].

Despite the establishment of such important principles so early on in the development of computer systems and programming languages, and in particular Tony Hoare's *Communicating sequential processes* (CSP) [15], they have since largely been ignored. This can be attributed to the advent of *integrated circuit* (IC) and *very large scale integration* (VLSI) techniques which combined with huge consumer demand for ever more performance-intensive software has resulted in processor design based on *frequency scaling*. That is, increasing the execution rate of sequential code, which has been a pragmatic and straight-forward approach for the designers of both hardware and software. The rate of this follows, or has been arguably dictated by Moore's law [26].

In the last few years however, frequency scaling has become increasingly difficult, and over the 30 years of systems design driven by it, only relatively small academic and commercial groups have been active in the areas of software concurrency and the design of parallel architectures. It is now accepted though that parallelism will be the primary method of improving system performance [1], but we don't currently know how to effectively architect or program parallel systems.

1.1 State of the art parallelism

Parallelism is now becoming pervasive in system design at many different levels and to varying degrees. Current state of the art high performance computing (HPC) systems employ the order of 10^5 processing cores and are central to the rapidly growing areas of computational science, utility computing and the Internet. Parallelism has also penetrated consumer markets with dual and quad core processors now the standard in desktop computers and laptops.

In spite of this, parallelism is still deployed in specific areas addressing particular requirements, with little regard to a more general approach. This is evident in the wide variety of recent CMP designs, e.g. [17, 18, 31] and HPC systems e.g. [5, 28]. The result of this is an increasing gap between the architectures and programming languages, and the application users writing the algorithms. For CMPs, the result is they are not easily amenable to *general-purpose* computing and instead have to focus on specific applications predominantly in the embedded space where parallelism is more explicit and hence easier to program. For HPC, the hierarchy of parallelism is increasing with growing system sizes and numbers of cores integrated onto a single chip.

These problems are highlighted by one of the key requirements of the *International exascale software project* (IESP), recognising the current complexity of systems and their underutilisation, which is to develop a new model of computation to establish standard ways of exploiting parallelism and system performance [8]. Emerging from this is a requirement to be able to more generally exploit parallelism. This work outlines language-level features to allow parallelism to be effectively exploited.

1.2 Universal parallel computers

The key idea behind the von Neumann architecture and reason for its success is that it provides an *efficient abstraction* from the implementation of different computer systems, enabling programs to be expressed at a high level and to be transportable between different platforms.

This idea is expressed more formally as *universality*, a concept introduced by Alan Turing in 1937 [32], stating that a computer may be viewed both as a special purpose device for executing a particular program, as well as a device capable of simulating all programs. It means that special purpose machines have no significant advantage, since general-purpose machines can perform the same functions almost as fast [34]. The consequence of building special-purpose parallel computers is the programmer must program with explicit consideration for the specialised architecture [33]. It is therefore crucial that a universality concept is developed for parallel computers so that concurrent programming languages hide irrelevant hardware detail and expresses parallelism concisely [11].

The most important elements in concurrent programming are *processes*, a sequence of computational steps, and *communication*, transfer of data between a set of simultaneously executing processes [11]. It has been observed that the concept of a *universal parallel computer*, a device enabling parallelism to be exploited effectively with high level programming languages, is essential to the success of parallel systems [33], and that the realisation of this concept depends on the provision of operations at the hardware level supporting these concurrency primitives. This paper examines several mechanisms supporting concurrency and discusses how they are implemented and used to express concurrent programs.

1.3 Support for concurrency

We are familiar in sequential computers with a hierarchy of dynamic resource allocation mechanisms, such as stack memory for procedure calls and garbage collection. Such issues are not irrelevant to the programmer, but he or she may choose at what level they are dealt with, through their choice of language and features within it, in order to represent their program clearly. The situation should be no different for parallel computers, where the resource is not just memory, but also processors.

The goal of continued increases in performance of hardware from parallelism now depends on the effectiveness of the *software* to harness it within a system. Realistically, programmers must be provided with language-level features aiding the problem of process-to-processor allocation.

It is vital that such features are provided as elements of a language so that they can be properly optimised by a compiler, which is not possible with library-based functionality, for example, with the Message Passing Interface (MPI) library standard for HPC, which allows programs to communicate between computers over a network.

This paper will show that the combination of the concepts of *process migration* and *parallel recursion* can provide this necessary support, and that they can be efficiently implemented with the support from the hardware, allowing the simple and elegant expression of concurrent programs.

1.4 Related work

There has been little work in the area of parallel recursion, and particularly with regard to implementations. Per Brinch Hansen founded many of the ideas with his work studying parallel programming paradigms [12], and design of languages, notably SuperPascal [14], which featured channel communication and parallel recursion [13], but its only implementation was interpreted.

More recently, the language occam-π [35], has been developed which has brought concepts of process and data mobility from Milner's π-calculus [25] to the occam language [19], which itself is based on the principles of CSP. Occam-π allows dynamic process configuration with parallel recursion, but its features are complex, making the possibility of an efficient and scalable implementation difficult. Regardless though, it is currently only targeted at conventional CPUs, which rules out efficient process creation and mobility. Other languages which include similar features do exist, e.g. [2, 9, 29], but all are designed for existing architectures that do not support concurrency properly.

Several languages aimed at HPC programming are under development, notably Cray's Chapel language [3] and IBM's X10 language [4]. Both support the concepts of process migration and spawning of parallel activities as key ingredients in the design of concurrent programs. This is significant given the companies standing in the HPC markets, and validates the approach described in this paper. Critically though, both are implemented on conventional *x86* architectures with communication provided by external library functionality, limiting their efficiency.

Worth noting as well are recent developments with general-purpose languages for graphics processing units (GPUs), such as CUDA [27] and OpenCL [30]. GPUs are inherently data-parallel and heterogeneous architectures, which make it difficult for the languages to be architecture-independent. They require unwieldy and complex library calls for initiation and termination, and are really only applicable to problems that are amenable to data-parallelism.

1.5 Language features supporting concurrency

In this section, the language features for parallelism and process-level communication are first introduced as they form the key components for the following explanations of process migration and parallel recursion. Example code is included in the following sections, which is written in a simple imperative language with keywords and syntax embodying these features. These features will be described as the accompanying ideas are introduced.

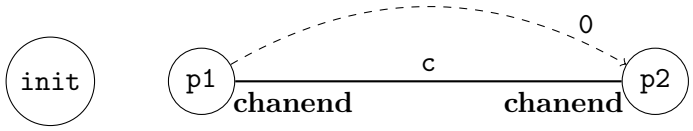
1.5.1 Parallelism & channel communication

Communication is provided at a language-level by abstract channel entities, through which values are transmitted. The idea of channel communication is already an established concept in languages such as Erlang [9], Go [29] and occam [19], all of which borrow from the ideas of CSP [15]. In the language used in the following examples, a channel is declared as an entity

of type **chan** and two **chanend**s of a channel are shared between processes. Input and output operators '`?`' and '`!`' are used to send and receive values bidirectionally along a channel. An input statement reads an incoming value from a `chanend` and assigns it to a variable. An output operation reads a value from an expression and sends it on the channel referenced by the `chanend` variable. For example:

```
proc init() is          proc p1(c: chanend) is      proc p2(c: chanend) is
  var c: chan;            var x: integer;             var y: integer;
{ p1(c) | p2(c) }       { x:=0 ; c!x ; c?x }        { c?y ; c!y+1 }
```

An initialising process (left) executes procedures `p1` and `p2` in *parallel*, by use of parallel composition denoted by the '`|`' separator, in contrast to '`;`', the sequential separator. Procedure `p1` then sends the value 0 to `p2`, which increments it by one and sends it back. The parallel composition is provided by threaded *fork-join parallelism*, where the running thread forks into two or more distinct threads of execution. On completion, the forked thread rejoins the original thread. Pictorially, we have the following communication structure:

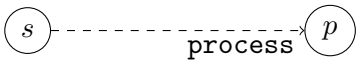


Channel output operations must be matched with a corresponding input operation, otherwise a deadlock situation will occur. For greater flexibility, channels are first class entities within the language, meaning that they can be passed as parameters, returned from a function or assigned to another channel reference, a concept established in π-calculus [25].

1.5.2 Process migration

Process migration allows computations to travel between distinct physical processors during execution. It is a mechanism by which computations on remote processors can be initiated and terminated efficiently. In contrast to conventional static process-to-processor mappings, it allows much greater flexibility and can be provided as a simple language feature with the **on** statement [20], for example:

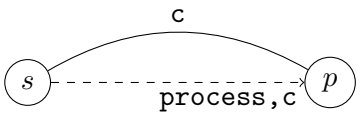
```
on p do process()
```



This statement causes the procedure `process` to be sent from the process running on processor s to be executed remotely on processor p.

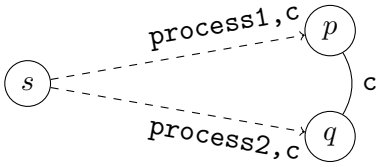
A migrated process could also take with it a **chanend** of and channel `c`, so that the controlling process can then communicate with it. In the example below, after initiating `process` on processor p, the sending process sends a `value`, which is received by the remote `process`.

```
var c: chan
{ on p do process(c)
; c ! value
}
```



An extension of idea allows an entire **chanend** `c` to be transmitted and used remotely to communicate between processes, a idea key to later constructions. For example:

```
var c: chan
{ on p do process1(c)
; on q do process2(c)
}
```



Here, `process1` and `process2` are executed in parallel on remote processors p and q, sharing the channel `c`.

The implementation of the **on** keyword involves transmitting to the remote node, a *closure* of the process, which is a complete description, including parameter values and communication channels, allowing it to autonomously execute, communicate and transmit back any result values. The movement of program and data is relatively straight forward, it must ensure correct moving or copying semantics are observed [24], and that any updated variables are removed from the cache. The protocol for communicating channels over channels is slightly more complex but is discussed in detail in [23]. In practise, this functionality would be realised as a small run time program executing on each processing core, able to transmit, receive, initiate and terminate process closures.

A process migration mechanism allows two important actions. The first is that processes can be moved to other processors in order to distribute computational load more evenly over a system. Secondly, when data is physically distributed across memories, it allows the movement of a process to operate on data *in situ*. As process closures will often be many orders of magnitude smaller than the data on which they operate, migration will be more economical than data movement. As the size of computer systems increases, process mobility will become increasingly important as it optimises the use of the interconnect and processors, maximising performance and minimising power [20].

1.5.3 Parallel recursion

Parallelism is a mechanism by which a computation is broken down in to a number of smaller ones that are performed simultaneously [11]. This is matched with the concept of recursion, where the solution to a problem depends on solutions to smaller instances of the same one, known as a *divide-and-conquer* approach. A finite *parallel recursive process* in theory allows the limitless generation of sub-computations and, when combined with a mechanism for process migration, will enable programs to effortlessly scale to an arbitrary number of cores[1].

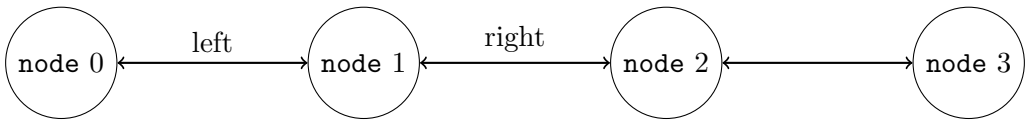
To illustrate the idea of parallel recursion, consider the following procedure which creates a set of processes organised in a pipeline communication structure:

```
proc pipe(i, len: val; left, right: chanend) is
  var middle: chan
  if i = len - 1 then node(i, len, left, right)
  else { node(i, len, left, middle) | pipe(i + 1, len, middle, right) }
```

When the index `i` is less than the length `len`, execution of the **par** statement causes a `node` procedure and recursive calls to `pipe` to be executed in parallel. When the index is equal to the length a single `node` procedure is executed, and the recursion terminates. For example, if the `pipe` procedure was called with a length of 4: `pipe(0, 4, nil, nil)`, then pictorially, the

[1]However, the size of any system is limited and the recursion may grow beyond it, at which point threads could be *virtualised* on the same core. Consequently, this could lead to reaching a core's memory capacity, where it may be necessary to terminate the program altogether.

process communication structure would have the following form, where the labelled edges show the channel names from the perspective of `node` 1.



This pipeline structure then allows values to be communicated sequentially among the nodes. For example, the following `node` procedure receives a value from its `left` channel (unless it is at index 0) with the channel input operator, and forwards it on through the `right` (unless it is at index `len`), channel using the output operator, adding its own index to the result `i` in the structure.

```
proc node(i, len: int; left, right: chanend) is
  var received: int
{ received := 0
; if i > 0   then left  ? received
; if i < len then right ! received + i
}
```

It is possible to express this kind of construction in existing languages that support process creation and channel communication, such as Erlang [9] or Go [29], but their implementations target conventional *x86* processors so they can only achieve thread-level parallelism, that is, time-multiplexed streams of sequential execution. It is possible with recent shared memory multi-processors to achieve true parallelism, for instance, Erlang's virtual machine supports symmetric multiprocessing, but this approach only offers limited scalability.

1.5.4 Process structures

Although the pipeline structure given in the example is seemingly simple, it is in fact significant and underpins a number of important parallel algorithms. Per Brinch Hansen's work in studying the parallel implementation of certain key algorithms in computational science, showed that many of them share the same *process structure* [12]. In particular, the pipeline can be used to implement algorithms solving systems of linear equations, primality testing, multiplication and n-body simulation (amongst others). His key insight in these cases was that each algorithm solves an instance of the all-pairs problem, where every possible subset containing two elements is chosen from n elements, which is efficiently implementable with a pipeline. Similar fascinating results were shown for trees, cubes and meshes. This work has recently been extended in [1].

1.6 Hardware support for concurrency

The language-level features described deal with *primitive* parallel operations, namely dynamic process creation and communication. It is essential for an efficient implementation of these features that *direct* support is provided by the hardware. In many parallel computer systems, these operations are implemented in software and are consequently orders of magnitude slower than primitive sequential operations such as memory accesses or sub-routine calls, for example communication protocols in HPC systems are implemented to some extent in software. In such systems it is difficult to provide these important mechanisms efficiently.

Equally, it is necessary that language features for concurrency are simple enough that they are capable of an efficient and *scalable* implementation.

2 Concurrent programming

When we have hardware to support both the concepts of parallel recursion and process migration, the result is both simple and powerful. HPC employs large-scale parallelism, but in general, most systems lack proper support for concurrency, especially with the growing hierarchy of parallelism, primarily at the multi-thread and multi-core levels. The result is that communication is provided as library functionality, most commonly by MPI implementations. MPI itself represents a limited state of the art, in which it is only economical to perform few, large communications, and although the MPI-2 standard supports dynamic process creation, it is not widely implemented or used. This approach forces an unintuitive mix of task and data parallelism.

With the parallel recursive approach suggested in this paper, the style of programming employed changes from the conventional focus on data structures to the concurrent concept of *process structures* [21], such as the pipeline structure shown previously. This allows a much more intuitive expression of concurrent programs. A simple but powerful example which combines process migration with parallel recursion to demonstrate this is the rapid distribution of processes across a system.

2.1 Rapid process spawning

In any parallel computer, distribution of processes over the network of physical processors is fundamental to the operation of the system. Using parallel recursion and dynamic movement of processes, we can use concurrency itself to efficiently and rapidly distribute concurrent computations across both *space* and *time* in a network.

Consider the following recursive procedure `d` to distribute the procedure `node` over `n` processors, an example borrowed from [20]:

```
proc d(t, n: int) is
  if n = 1 then node(t)
  else { d(t, n/2) | on t + n/2 do d(t + n/2, n/2) }
```

It works by offloading a copy of itself to a remote processor each time it recurses. These offloaded processes then, themselves, continue this behaviour. Each recursion sees a doubling of the capacity to initiate computations, in the structure of a binary tree. When each instance of `d` executes with `n` equal to 1, it executes the `node` procedure, halting the recursion. The parameter `t` indicates the node identifier. For the execution of `d(0, 8)`, which distributes `node` over 8 processors, $p_0, p_1, ..., p_7$, the behaviour is illustrated by the following table, showing at each time step the state of each processor, given by the procedure it is executing.

Step	p_0	p_1	p_2	p_3	p_4	p_5	p_6	p_7
0	d(0,8)							
1	d(0,4)				d(4,4)			
2	d(0,2)		d(2,2)		d(4,2)		d(6,2)	
3	d(0,1)	d(1,1)	d(2,1)	d(3,1)	d(4,1)	d(5,1)	d(6,1)	d(7,1)
4	node(0)	node(1)	node(2)	node(3)	node(4)	node(5)	node(6)	node(7)

The time steps here are given by each recursive call, but to maximise the speed at which computations can be distributed, the receiving node should immediately start executing the incoming program so that a next recursive call and process movement can be performed as quickly as possible. Intuitively, you could think of these processes as travelling like 'viruses' through the network. In general though, the time required to initiate a computation on a set of

n processors is $O(\log n)$: an exponential growth rate that will populate even very large systems quickly.

The procedure `d` is powerful as it allows the set of distributed processes to be connected in a process structure that may be matched to that of the underlying network, or to any other of arbitrary complexity.

2.2 A new approach?

It is striking from the rapid process spawning example how parallel recursion and process migration can be used to express such a mechanism so simply. Conventional parallel architectures and languages provide only restricted means for process creation, influencing the way in which concurrent programs are written and consequently system performance leveraged. Providing such fine control over the dynamic creation and termination of processes with the features described allows much better control of this.

It is well known that there is a large gap between the peak performance of HPC systems and the actual performance of today's algorithms, this is attributable to the styles of programming used. In particular, it is common for a data-parallel approach to be taken where a specific process is executed concurrently on the same or different sets of data, such as in a ray tracer where the colour of each pixel can be calculated independently using the world model. During execution, all processes synchronise after each frame is rendered and it is likely that many of the processes will become idle when the rays it is calculating make no further intersections. It must then wait for the process calculating the most complex ray to complete. With an efficient means of dynamic process creation, it would be easy to move away from this synchronous data-parallel approach to better utilise the spare capacity in the system.

3 A real implementation

This paper suggests that we need language-level features supporting the expression of concurrent programs and that they must be supported properly by the hardware. The key features introduced relate to the primitive parallel operations of process creation, mobility and communication. The concepts of parallel recursion and process migration have been known for a long time and exist in several implementations, but have not been realised in the context of a sympathetic architecture.

As part of this work, we intend to realise these concepts by an implementation on a state of the art parallel processor, and for this we have identified the XMOS XCore processor. This architecture is general-purpose, scalable and has been designed from the ground up to support concurrency [22]. In particular, it includes specific instructions for channel communication and fork-join parallelism, allowing concurrent language features to be built directly on top of the hardware. Its existence shows that it is possible to efficiently implement parallel recursion and process migration mechanisms.

We have written a complete bespoke compiler as preliminary work towards this goal. It implements as a basis, a small language (the one used in the examples in this paper), with features sufficient to properly demonstrate the mechanisms, and is targeted at the XCore architecture. To enable parallel recursion at a thread level, a special stack allocation scheme will be employed to allow the formation of a tree-structured stack such as the one Per Brinch Hansen described in [13]. The implementation of a process migration mechanism is more complex and is divided into two components. The first of these is run-time functionality, which will be resident on each core as a light-weight kernel. Each core's kernel will be able to mediate the transmission, delivery and execution of process closures between cores in the network. The second component

is the translation by the compiler of the `on` statement into a sequence of instructions able to interact with the kernel and instruct it to migrate a processes to a specified processor.

We intend the full implementation comprising the features in the language and accompanying run-time to follow in due course[2].

4 Conclusion

This paper has developed the concepts of process migration and parallel recursion as features of a programming language and shown how they can can be used as key general-purpose constituents in the design of concurrent programs. In particular, it has illustrated how parallel recursion can be used to elegantly express process structures, and combined with process migration, can be used to rapidly distribute a computation over a set of processors, an ability key in the utilisation of systems employing large-scale parallelism.

Integral to the efficient implementation of these features is support for underlying primitive parallel operations of process creation, mobility and communication in hardware. The XMOS XCore architecture is proof of their existence, and an excellent platform on which they can be demonstrated. Initial exploration of this approach has been promising and symbiotic. The results of an implementation will illustrate the effective and efficient operation of these mechanisms, which will in terms of expressiveness and performance, be unmatched by any current systems.

References

[1] Asanovic, Bodik et al. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[2] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: an efficient multithreaded runtime system. *SIGPLAN Not.*, 30(8):207–216, 1995.

[3] B.L. Chamberlain, D. Callahan, and H.P. Zima. Parallel programmability and the Chapel language. *Int. J. High Perform. Comput. Appl.*, 21(3):291–312, 2007.

[4] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 519–538, New York, NY, USA, 2005. ACM.

[5] Cray. ORNL's Jaguar XT5 Supercomputer. `http://www.cray.com/Products/XT/ORNLJaguar.aspx`. Accessed April 2010.

[6] Edsger W. Dijkstra. The structure of the "THE"-multiprogramming system. *Commun. ACM*, 11(5):341–346, 1968.

[7] Edsger W. Dijkstra. Cooperating sequential processes (1965). pages 65–138, 2002.

[8] Dongarra, Beckman, et al. International exascale software project roadmap. Technical report, January 2010.

[9] Erlang programming language. `http://www.erlang.org/`. Accessed April 2010.

[10] Per Brinch Hansen. Rc 4000 software: multiprogramming system (1969). pages 153–197, 1969.

[11] Per Brinch Hansen. The nature of parallel programming. *Natural and Artifical Parallel Computation*, pages 31–46, 1990.

[2] More information about the status of the project and implementation source code will be made available at `www.cs.bris.ac.uk/~hanlon/language`

[12] Per Brinch Hansen. *Model programs for computational science: parallel programming paradigms*. John Wiley and Sons, Ltd, 1993.

[13] Per Brinch Hansen. Efficient parallel recursion. *SIGPLAN Not.*, 30(12):9–16, 1995.

[14] Per Brinch Hansen. SuperPascal: a publication language for parallel scientific computing. pages 495–524, 2002.

[15] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978.

[16] C. A. R. Hoare. Towards a theory of parallel programming (1971). pages 231–244, 2002.

[17] IBM Research. The Cell project. `http://www.research.ibm.com/cell/`. Accessed March 2010.

[18] Intel's Tera-scale Computing Research Program. Single-chip cloud computer. `http://techresearch.intel.com/articles/Tera-Scale/1826.htm`. Accessed March 2010.

[19] David May. Occam. *SIGPLAN Not.*, 18(4):69–79, 1983.

[20] David May. *The Transputer revisited*. Millennial Perspectives in Computer Science: Proceedings of the 1999 Oxford-Microsoft Symposium in Honour of Sir Tony Hoare. Palgrave Macmillan, 1999.

[21] David May. Invited talk 1- past, present, and future communicating processors. In *NOCS '08: Proceedings of the Second ACM/IEEE International Symposium on Networks-on-Chip*, page xiv, Washington, DC, USA, 2008. IEEE Computer Society.

[22] David May. *The XMOS XS1 Architecture*. XMOS Ltd., 10 2009.

[23] David May and Henk Muller. A simple protocol to communicate channels over channels. In *Euro-Par '98: Proceedings of the 4th International Euro-Par Conference on Parallel Processing*, pages 591–600, London, UK, 1998. Springer-Verlag.

[24] David May and Henk Muller. Copying, moving and borrowing semantics. In Alan Chalmers, Majid Mirmehdi, and Henk Muller, editors, *Communicating Process Architectures – 2001*, pages 51–62. IOS Press, September 2001.

[25] Robin Milner. *Communicating and Mobile Systems: the Pi-Calculus*. Cambridge University Press, June 1999.

[26] G. E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8):114–117, April 1965.

[27] NVIDIA. CUDA Zone. `http://www.nvidia.com/object/cuda_home_new.html`. Accessed April 2010.

[28] SGI. SGI Altix UV. `http://www.sgi.com/products/servers/altix/uv/`. Accessed April 2010.

[29] The Go programming language. `http://www.golang.org/`. Accessed April 2010.

[30] The Khronos group. OpenCL - The open standard for parallel programming of heterogeneous systems. `http://www.khronos.org/opencl/`. Accessed April 2010.

[31] Tilera. Tile64 processor family. `http://www.tilera.com/products/TILE64.php`. Accessed April 2010.

[32] A. M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proc. London Math. Soc.*, s2-42(1):230–265, January 1937.

[33] L. G. Valiant. Optimally universal parallel computers. pages 17–20, 1988.

[34] L. G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, 1990.

[35] P.H. Welch and F.R.M. Barnes. Communicating mobile processes: introducing occam-pi. In A.E. Abdallah, C.B. Jones, and J.W. Sanders, editors, *25 Years of CSP*, volume 3525 of *Lecture Notes in Computer Science*, pages 175–210. Springer Verlag, April 2005. to appear.