# Multicore Scheduling for Lightweight Communicating Processes

Carl G. Ritson, Adam T. Sampson, Frederick R.M. Barnes

*School of Computing, University of Kent, Canterbury, Kent, UK*

**Abstract**

Process-oriented programming is a design methodology in which software applications are constructed from communicating concurrent processes. A typical process-oriented design involves the composition of a large number of small isolated component processes. These concurrent components allow for the scalable parallel execution of the resulting application on both shared-memory and distributed-memory architectures. In this paper we present a runtime designed to support process-oriented programming by providing lightweight processes and communication primitives. The runtime's scheduler, implemented using lock-free algorithms, automatically executes concurrent components in parallel on multicore systems. Heuristics dynamically group processes into cache-affine work units based on communication patterns. Work units are then distributed via wait-free work-stealing. Initial performance analysis shows that, using the algorithms presented in this paper, process-oriented software can execute with an efficiency approaching that of optimised sequential and coarse-grain threaded designs.

*Key words:* Concurrency, Multicore, Process-oriented, Scheduling

## 1. Introduction

Interest in concurrent programming techniques is growing as a result of the increasing ubiquity of multicore systems on the desktop, and in mobile and embedded systems. Designing applications which scale to current and, more importantly, future multicore systems is an important research topic.

---

*Email addresses:* cgr@kent.ac.uk (Carl G. Ritson), ats@offog.org (Adam T. Sampson), frmb@kent.ac.uk (Frederick R.M. Barnes)
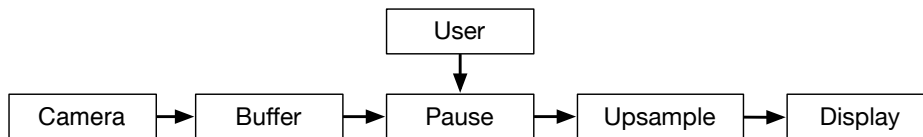
Figure 1: Pausable camera display, process-oriented pipeline example.

Process-oriented programming is one concurrency paradigm available for creating such scalable software.

Process-oriented programming employs concurrency as a design tool for constructing software applications. Small independent concurrent processes are composed to form larger components, which through continued composition form the application as a whole. The developer is dissuaded from using forms of sharing which may introduce race-hazards and aliasing errors; they may even be prevented from doing so by the compiler [1]. Instead, interaction between processes takes place via explicit communication and synchronisation primitives. These expose dependencies at the design level and permit diagrammatic representations such as Figure 1. While in the past message-passing designs mapped processes one-to-one to processors [2, 3], process-oriented designs are intended to be architecture independent [4–7].

Parallel execution potential is inherent in a process-oriented design, and is bounded only by the number of ready processes. While the size of components varies with the design style chosen, a typical process-oriented design can have thousands of processes. Furthermore, as processes are created and connections between them made at runtime, truly dynamic systems can be modelled directly as process networks [8, 9]. The explicit transfer of state using communication allows unmodified designs to be serialised for a single processor [10], and parallelised across shared-memory and distributed-memory multiprocessor systems [11, 12].

Process-oriented software needs to execute with comparable performance to a sequential implementation in the absence of hardware parallelism, and automatically scale when multiple processors are available. To make this possible, scheduling and communication overheads must be minimised. Communication between processes must have an overhead comparable to calling a procedure, or invoking a method on an object. Runtime implementations which build communication upon common locking and operating system synchronisation primitives do not provide sufficient performance. Process-oriented software also requires functionality not provided by many lightweight threading frameworks (see section 6).

2

In this paper we present implementation details of our runtime kernel for realising scalable process-oriented programming on multicore systems. Specifically we contribute:

- Wait-free algorithms for process migration via work stealing [13, 14].

- Automatically grouping communicating processes into cache-affine work units at runtime.

- Multiprocessor-aware interprocess communication with an average overhead of only 140 cycles on modern commodity hardware.

- A mechanism for choice over a set of communication channels inspired by that of the INMOS Transputer [15, 16], made multiprocessor safe.

Our runtime is a C library and provides a C API. It can also be used through occam-$\pi$, a concurrent programming language which supports process-oriented design [9, 17]. The occam-$\pi$ language extends original occam [6] with channel and data mobility. It is rooted in the formalisms of Hoare's CSP [10, 18], and Milner's $\pi$-calculus [8].

occam-$\pi$ is being used as an implementation language for complex systems and operating systems research [19, 20]. A complex system can be modelled as agents, each of which is a composition of concurrent processes. Agents move through and interact with a simulated environment using communication and dynamic changes in connectivity. The environment itself is also a composition of concurrently executing processes. Simulations can scale up to hundreds of thousands of processes [21]. Using the runtime presented in this paper, these simulations can be executed in real time on commodity workstations, utilising all processing resources available.

The rest of the paper is as follows. In section 2 we introduce our lightweight processes and a system for scheduling them across multiprocessors while attempting to enhance cache utilisation. Section 3 describes communication channels, which can be used to pass information between processes executing on the same processor or separate processors in a shared-memory system. Primitives for choice, protecting shared resources and synchronising large numbers of processes are discussed in section 4. Finally an evaluation of performance comparing a sample set of applications implemented using other concurrency frameworks is presented in section 5. Related work is presented in section 6. Our conclusions and details of possible future work are in sections 7 and 8.
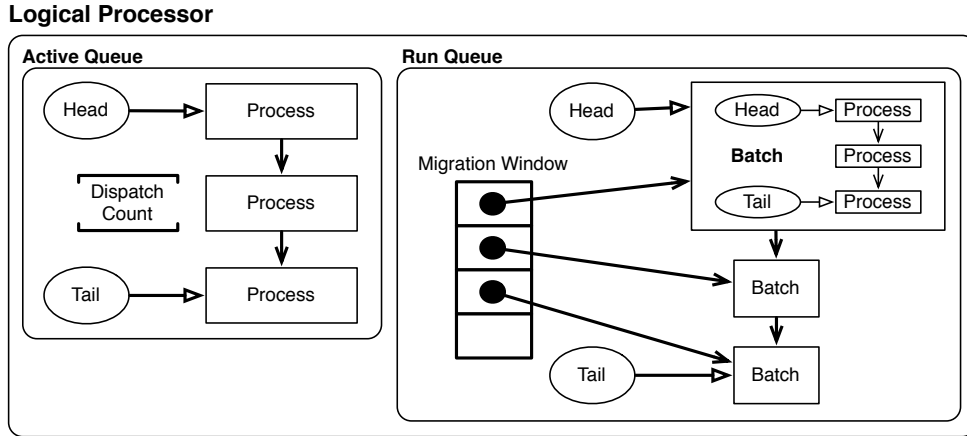
**Logical Processor**



Figure 2: A logical processor instance schedules batches of processes on each physical processor. Batches in the migration window can be stolen by other logical processors.

## 2. Processes

In this section we describe our runtime's cooperative scheduling model for concurrent processes. As the fundamental building blocks of process-oriented software, processes must be lightweight. Our design is intended to minimise context-switch times and memory usage, as well as exploit cache-affinity and hardware parallelism.

For reference in later sections we must first describe how processes are represented by the scheduler kernel. Each process has a *process descriptor* used to store state when descheduled or performing certain kernel calls. The descriptor can be allocated statically on the process stack, or when state does not need to persist across kernel calls it may be allocated dynamically at the point of call. In either case, the size of the process descriptor is eight machine words (32 bytes on a 32-bit machine). This minimal memory overhead makes the creation of very large numbers of processes practical.

The process descriptor contains the following elements:

```
        Alternation State        Priority and Affinity Mask
  Communication Data Pointer      Stored Instruction Pointer
        Queue Link Pointer          Stored Stack Pointer
```

*2.1. Scheduling*

Our scheduling model is divided into uniprocessor and multiprocessor components. In this subsection we focus on uniprocessor scheduling. We

also explain how processes can be grouped to enhance cache-affinity (reduce interprocessor cache communication).

For each physical processor in the host system a scheduler instance, a *logical processor*, is started. The logical processor contains a *run queue*, which is a linked list of *batches*. Batches are in turn linked lists of process descriptors, linked using the `Queue Link Pointer` field. An overview of this structure can be seen in Figure 2.

The scheduler executes each batch by moving the processes it contains to its *active queue*. A *dispatch count* is calculated based on the number of processes in the batch (multiplied by a constant) and bounded by the *batch dispatch limit* (constant). The dispatch count is decremented each time a process is taken from the active queue and executed. When the dispatch count reaches zero, and the active queue is not empty, the current active queue is stored into a new batch which is added to the end of the run queue.

### 2.1.1. Batching

As outlined above, batches are the base unit of work stored in scheduler data structures, and also for migration (see section 2.2). Batches address the issue of cache thrashing which can occur with process-oriented designs. It is highly probable that with a large number of processes switched frequently, the working set will exceed the processor's cache size. Processes and their data will be drawn into cache only to be rapidly evicted again, serving few or no hits. Modern processor architectures rely on cache to compensate for the high-latency of system memory, so sidelining the cache will severely restrict performance. The solution is to reduce the size of the working set by minimising the memory overheads on processes and partitioning the run queue. Vella proposed and experimented with dividing the run queue into batches of processes [11, 13]. Each batch is executed multiple times before moving on to the next. Relatively small batches fit well within the processor cache. Successive executions permit cache utilisation, thus improving performance.

Our scheduler attempts to group processes into the same batch when they communicate or synchronise with each other. By forming batches in this way, processes which communicate frequently are scheduled on the same processor, reducing interprocessor traffic. This is an improvement to Vella's techniques which used fixed-size batches determined by the developer and compile-time analysis. Our variable-size batches are formed and split automatically using runtime heuristics.
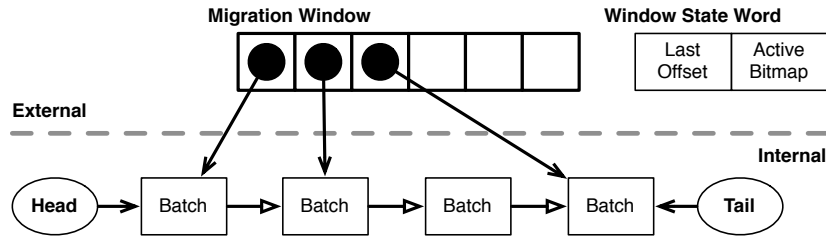
Figure 3: A fixed-size migration window array allows one logical processor to "steal" batches from another. This relates to the *Run Queue* in Figure 2.

Following a context switch, if the dispatch count is not zero, then the next process on the active queue is dispatched. Otherwise the scheduler restarts with a new batch. Context switches occur under two conditions. Most commonly, the current process blocks on a communication or synchronisation primitive and is descheduled. Alternatively, a process may cooperatively yield to the scheduler, in which case it is placed at the end of the active queue. With the exception noted in section 2.1.2, processes rescheduled (unblocked) by the currently executing process, for example by the completion of communication, are also placed on the end of the active queue. It is this action which draws related processes into the same batch.

### 2.1.2. Batch Size

If processes are always drawn into a batch during creation and communication, then one batch will eventually grow to encompass all processes in the system. This will prevent batching from having caching benefits as the working set will contain all active processes. Therefore a mechanism is required to prevent batches growing too large and to separate processes which lose association.

We observe that in high valency subgraphs of a process-oriented program network, there will be points when only one process in the subgraph is active. This process reschedules other processes in the subgraph which may then in turn become the only active process. Based on this observation we state that if while executing a batch there is a point at which only one process is active then that batch is probably optimal, i.e. contains only one subgraph. Conversely batches which never meet this condition during execution should be *split*. Batches are split by placing the head process of the active queue in one batch, and the remainder in another. This is a unit-time operation, and so can be carried out frequently. Repeated execution and split cycles quickly reduce large and unrelated batches to small related

6

process subgraphs. Erroneous splits will quickly reform based on the other scheduling rules.

Additional mechanisms to control batch size can be introduced by modifying the dispatch count in response to specific events. Process creation is one example. During process creation the new process is placed on the end of the active queue. Process creation does not cause a context switch; however, the runtime kernel decrements and tests the dispatch count. This prevents the batch size exceeding the dispatch count. Furthermore, if the dispatch count reaches zero and the aforementioned conditions for batch splitting are met, then a process creating many new processes will be split into a separate batch from the newly created processes. The newly created batch is then free to migrate (see section 2.2). Thus a process spawning a large number of children may continue to execute while its children begin execution on other logical processors in the system.

### 2.2. Process Migration

In this section we describe how logical processors interact as part of a multiprocessor system. In particular, we give details of our algorithms for wait-free work stealing.

Amdahl's law [22] states that for a *fixed* problem size, the total parallel speed up is limited by the sequential overhead. Hence when scheduling large numbers of processes on a multicore system, a single locked run queue represents a scalability bottleneck [23]. For this reason we do not use a global run queue in our runtime design.

Work is distributed between logical processors via migration. Processes are free to migrate between logical processors, except where restricted by an explicit affinity setting. Migration occurs in two circumstances:

1. A process which blocks during communication or synchronisation and is descheduled on one logical processor can be rescheduled by a process executing on a different logical processor. Unless prohibited by an affinity setting, the rescheduled process continues execution on the rescheduling logical processor.
2. A logical processor which runs out of batches to execute may *steal* batches from other logical processors [13, 14].

The first case occurs as part of the communication and synchronisation algorithms outlined in sections 3 and 4. The second case is the mechanism

7

1. Link the batch into the run queue linked list.
2. Load the window state word (see Figure 3).
3. Generate a new offset by incrementing the last offset, handling roll over where appropriate.
4. Record the generated offset into the batch data structure.
5. Atomically swap the batch pointer with the window entry at the generated offset.
6. If the result of the swap is not null, then a batch has been knocked out of the window; clear its stored offset to indicate it is no longer part of the window.
7. Update the window state word with the generated offset and active bitmap. This update is done with a blind write, and thus may overwrite updates from external dequeues.

Figure 4: Migration window local enqueue algorithm.

by which work is spread across the system. It is further underpinned by the observation that independent long-running subgraphs of processes will tend to be split into separate batches, which can be stolen by idle logical processors.

The run queue of each logical processor is private and cannot be accessed by other scheduler instances. To allow batch migration, a fixed-size window onto the end of each run queue provides access to other logical processors. The fixed size of the window allows it to be manipulated using wait-free algorithms [24, 25]. These provide freedom from starvation and bounded completion when contention arises, improving scalability over locks.

Lock-free and wait-free algorithms are often complex to implement and rely on expensive atomic memory operations such as *compare-and-swap* [26]. Despite this, efficient lock-free algorithms are more scalable than their lock-based counterparts [27]. The scalability of these algorithms motivated the decision to refine existing wait-free work-stealing for use in this work [13, 14].

Figure 3 shows the relationship of the migration window to the run queue. There are three algorithms for accessing the migration window: local enqueue, local dequeue, and remote dequeue. These algorithms assume a total store order memory model [28].

1. Remove the head batch from the run queue linked list.
2. If the batch has no stored window offset then the dequeue is complete (the batch is not in the window).
3. Atomically swap null with the migration window entry associated with the batch.
4. If the result is null then the batch has been *stolen* by an external scheduler. It is placed on a *laundry queue* on the logical processor for later cleanup. Dequeue of this batch fails, and we must restart the algorithm at step 1.
5. The bitmap in the window state word is updated to clear the associated bit. As with the enqueue algorithm, this occurs via a blind write.

Figure 5: Migration window local dequeue algorithm.

### 2.2.1. Local Enqueue

Figure 4 shows the algorithm used to place a batch onto the run queue of a logical processor and make it visible in the migration window.

Internal operations on the window will be more common than external operations, hence the presented algorithms are optimised for the uncontended case rather than the contended case. The effect of this optimisation is that the final step of the algorithm can produce corruption of the window state word. In the event of corruption the window will appear to external logical processors to contain more batches than it does; however, this does not affect correct operation of the external dequeue algorithm (only its operating efficiency). The result is an algorithm with a deterministic execution time and only one expensive atomic operation.

### 2.2.2. Local Dequeue

To dequeue a batch from its run queue, a logical processor uses the algorithm in Figure 5.

While the dequeue algorithm may fail and have to restart, it is bounded by the number of batches enqueued on the logical processor. In the worst case, every batch may have been stolen and the scheduler must scan every batch to discover this. Local scanning does not, however, create direct contention with other logical processors. Although processors may still contend for underlying system resources such as the memory bus.

1. Load the window state word, creating a local copy.
2. Rotate the active bitmap by the last offset.
3. Scan the bitmap to select an entry to steal. If the bitmap is empty, migration fails.
4. Atomically swap the window entry with null.
5. If the result is null, clear the associated bitmap bit (local copy) and restart at step 3.
6. Atomically clear the window state word bitmap bit; dequeue succeeds and the result of the atomic swap is the stolen batch.
7. A local copy of the stolen batch is created, and the original batch marked clean and its reference discarded.

Figure 6: Migration window remote dequeue algorithm.

### 2.2.3. Remote Dequeue

When one logical processor attempts to steal work from the migration window of another, it does so using the algorithm in Figure 6. This algorithm requires only two atomic operations in the optimal case.

Having migrated a batch the logical processor copies the contents to a new local batch data structure and marks the original batch as clean and discards the pointer to it. The originating logical processor will later collect the original batch structure and reuse it. This allows each logical processor to maintain its own pool of batch structures, and minimises cache ping-pong (inverting the scheme creates higher cache traffic).

## 3. Communication

Interprocess communication is central to process-oriented programming, for sharing state and synchronising computation. The efficiency of communication therefore directly affects the performance of process-oriented designs.

Our runtime kernel provides a single basic communication primitive for processes to exchange data: point-to-point synchronised channels. Synchronised channels require no buffers and data is copied or moved (depending on the mode of operation) directly between the source and destination processes. Buffered channels can be constructed efficiently by placing buffer processes between communicating processes. Transactions involving many parties sharing a channel are implemented by associating the channel with a mutual exclusion lock (see section 4).

1. Read the channel word.
2. If it is null or the alternation bit is set (the other party is waiting on multiple channels):
   (a) Store the process state in the process descriptor (instruction pointer, etc).
   (b) Store the destination or source buffer pointer in the process descriptor (`Communication Data Pointer`).
   (c) Atomically swap the process descriptor with the channel word.
   (d) If the result is not null, and the alternation bit is not set, then the read at step 1 was stale; jump to step 3.
   (e) If the alternation bit is set on the result, then trigger the event (see section 4.1.6).
   (f) A context switch occurs and a new process to execute is selected as described in section 2.
3. The channel word is not null, hence a process is blocked on it.
4. Load the destination or source buffer pointer from the blocked process descriptor.
5. Copy data or move references and transfer ownership.
6. Reset the channel word to null.
7. Reschedule the process blocked on the channel.

Figure 7: Channel communication algorithm.

Operations for channel input and output take a source or destination buffer and a size in bytes to copy. Alternatively the source and destination may be a reference to a memory object allocated through the runtime kernel, in which case the reference is moved between the processes together with ownership of the object.

A channel is represented by a single machine word. The word stores a pointer to the process descriptor (section 2), a structure guaranteed to be word-aligned. The lowest order bits of the word also carry state information about the process descriptor. For the algorithm which follows only the *alternation bit* is relevant. It indicates whether the process descriptor stored in the channel is *blocked* on this channel or *waiting* on a number of channels and events (see section 4.1).

Basic channel communication, regardless of direction, is performed using the algorithm in Figure 7. Using this algorithm the second process to reach

the channel completes the synchronisation and thus the communication. This results in, typically, only one of the two processes performing an expensive atomic operation.

## 4. Synchronisation

In addition to communication, processes often need to synchronise in ways which do not involve data exchange. This section describes additional synchronisation primitives supported by our runtime.

### 4.1. Alternation

For many purposes, blocking channel communication is sufficient; however, processes often need to multiplex between a number of channels and other events. Our runtime kernel supports choice over a number of channels and timer events: we call this *alternation*. occam-$\pi$ supports this via an `ALT` language construct.

Alternation allows a process to wait for one of a set of channels to become ready. When an element of the waited set becomes ready, the process is rescheduled and can make a choice as to which channel to communicate with. This is similar to the POSIX `select` system call.

In this section we present algorithms designed for one process waiting on a set of exclusively input or output channels, while other processes sharing those channels commit. This constraint is enforced by the present version of the occam-$\pi$ language and inherited from the original occam language. More general synchronisation algorithms are part of our ongoing research.

The following subsections describe, in order, the steps involved in alternation.

### 4.1.1. Initialisation

The `Alternation State` field of the process descriptor is initialised. The alternation state consists of:

- *flags* indicating what stage of alternation the process is in. The initial flags are `enabling` and `not ready`.

- a *reference count* which tracks the number of pointers to the process descriptor, initially one. When a logical processor triggers an event which is part of an alternation it takes one of these references. The alternation only completes when all references have been counted back through the disable algorithm or via event triggers.

12

1. Read the channel word.
2. If the channel word is not null, then atomically clear the `not ready` flag of the alternation state. The enable operation completes indicating the channel is ready.
3. Atomically swap a pointer to the process descriptor with the *alternation bit* set into the channel word.
4. If the result is not null, then the value in step 1 was stale. Write the result back to the channel word and continue as in step 2.
5. Atomically increment the alternation state reference count.

Figure 8: Channel enable algorithm.

1. Read the channel word.
2. If it does not contain a pointer to the process descriptor of the alternating process, then the channel is ready. The operation returns indicating the channel is ready.
3. Atomically compare-and-swap null to the channel, if this fails then the channel just became ready; the algorithm completes as in step 2.
4. Channel is not ready, decrement the reference count in the `Alternation State`.
5. Return value indicates channel not ready.

Figure 9: Channel disable algorithm.

1. Read the reference count of the `Alternation State`.
2. If the reference count is one then alternation is finalised; leave the algorithm.
3. Save the process state as if to context switch.
4. Atomically decrement and test the reference count.
5. If the reference count does not reach zero then context switch.

Figure 10: Alternation finalisation algorithm.

### 4.1.2. Channel Enabling

Each channel a process alternates over is enabled using the algorithm in Figure 8.

### 4.1.3. Waiting for Events

Once the process has enabled all the events it makes a kernel call to wait. An atomic compare-and-swap is used to clear the `enabling` and `not ready`

flags, and set the `waiting` flag. If the compare-and-swap succeeds then the process is descheduled and a context switch occurs. Failure indicates that an event has become ready, in which case the `enabling` flag is atomically cleared and execution of the process continues.

### 4.1.4. Channel Disabling

Having been woken up, the process disables channels using the algorithm in Figure 9.

### 4.1.5. Finalisation

Having disabled all channels, the alternation is finalised using the algorithm in Figure 10. This completes the alternation and communication with any ready channels may take place.

### 4.1.6. Event Trigger Algorithm

Logical processors execute the *event trigger algorithm* (Figure 11) to signal an alternating process that one of its waited events has become ready. This is the algorithm referenced at step 2(e) of the basic channel communication algorithm in Figure 7. If a logical processor in the event trigger algorithm holds the final reference then it is responsible for rescheduling the alternating process.

### 4.2. Mutual Exclusion

Section 3 describes communication channels capable of synchronous point-to-point exchanges involving pairs of processes; however, there are designs which require multiple communication peers to use the same channel. This is particularly useful for implementing the deadlock-free client-server design

pattern [29], in which a number of clients communicate with a single server over channels.

To support this functionality mutual exclusion locks can be associated with the channel directions. This allows ordered multi-access (used by multiple processes) channels to be constructed. The lock *claim* and *release* algorithms are non-blocking and prevent starvation using FIFO queuing. Importantly, the occam-$\pi$ compiler enforces claim and release semantics on these locks, so that an application developer cannot forget to release the channel lock.

*4.3. Barriers*

Our runtime also supports a barrier synchronisation type. Processes can *enroll*, *resign* and *synchronise* on such barriers. Processes synchronising on a barrier are blocked until all other processes enrolled on the barrier are also synchronising. Barriers may also be communicated by reference over channels, atomically enrolling the receiver as part of the communication; this permits semantics such as those described by Welch and Barnes [30].

Barriers of this type are useful in implementing agent simulations. Each agent is enrolled on a barrier and synchronises on it to maintain time-step with the other agents in the simulation. With many thousands of agents synchronising, the performance of barrier operations is critical. It is also important to minimise the time between barrier completion and returning to the state where all enrolled processes are scheduled for execution across available logical processors.

## 5. Performance

In this section we present preliminary results from a number of benchmarks we have developed to test and compare the performance of our runtime. The source codes for these benchmarks are publicly available [31].

All our benchmarks were performed on an eight core Intel Xeon workstation composed of two E5520 quad-core processors running at 2.26GHz. Each core has two hardware threads and 256KiB of L2 cache, giving a total of 16 hardware threads and 2MiB L2 cache. Each processor also has 8MiB of L3 cache and an independent memory bus, creating a non-uniform memory architecture when both cores from both processors are used. For all tests the workstation ran Linux 2.6.30. Where appropriate, the `taskset` utility and runtime flags were used to restrict the cores on which benchmarks ran.

15

Unless otherwise noted only one hardware thread was used per core, and cores in the same processor were selected in preference to cores in separate processors. For example, benchmark results for five cores will involve four cores from one processor and one core from the other processor.

Comparison of our results was performed by close reimplementation of our benchmarks using multiple languages and concurrency frameworks:

- *CCSP C* – our runtime programmed using its C API.

- *CCSP occam-π* – our runtime programmed using the occam-π.

- *Erlang* – a functional programming language with asynchronous message passing [1]. We used version 5.7.3 with HiPE [32].

- *Haskell* – a functional programming language with lightweight threads and one-place buffered channels provided by the `MVar` primitive. We used GHC version 6.8.2 [33].

- *Java* – benchmarks use primitives from `java.util.concurrent`, in particular `ArrayBlockingQueue`. We used OpenJDK version 1.6.0_0_b11.

- *pthread C* – POSIX threads accessed via the GNU C library. Mutual exclusion (`pthread_mutex_t`) and condition variables (`pthread_cond_t`) are used to construct one-place buffered communication channels.

*5.1. Mandelbrot*

In this first benchmark, we test the coarse-grain scalability of our runtime by calculating iterations of the mandelbrot set using a group of worker processes. Our example calculates 128 frames or iterations. A central process farms out lines of each iteration to 128 worker processes. The workers return the calculated image lines to the central process.

Areas of the mandelbrot set can be independently computed in parallel, hence execution speed should increase linearly as more processor cores are utilised. The results in Figure 12 show the our runtime scheduler performs this parallelisation correctly.

In addition to enabling all cores, the results show further scaling to all hardware threads (not actual physical processors). Surprisingly, our

---

[1]We have not forced synchronised communications, but instead we coerced our designs to function with asynchronous messaging. This should be a performance benefit for Erlang.
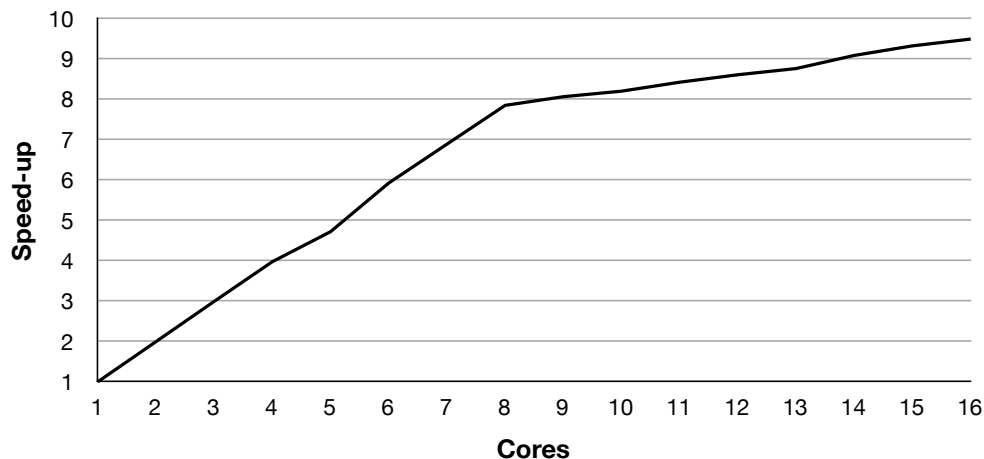
Figure 12: Scaling performance of mandelbrot set render with increasing numbers of hardware threads.

runtime scheduler is able to extract further parallelism from these threads without performance decrease. This suggests the hardware threads are more computationally capable than expected.

*5.2. Process Ring*

To examine communication overheads, we construct a ring of *n element* processes, and one *initiator* process. Element processes loop: they receive an integer token from the previous process in the ring, increment it, then send it on to the next process. The initiator, adds tokens, counts them passing and after a given count removes them from the ring. By increasing the number of tokens "in flight" around the ring, we increase the number of potentially concurrently executing processes.

Given the time taken for a single token to circulate the ring we can estimate the average communication time of each language runtime as $time \div ((elements + 1) \times roundtrips)$. For all our examples, there are 255 element processes and tokens make 1024 round trips. With 255 elements it is likely that all processes will fit within the processor caches, allowing us to examine the best-case communication time.

Table 1 shows communication times in nanoseconds. These are based on the circulation of a single token with one core or eight cores enabled.

The communication times for Erlang and our runtime are relatively unaffected by the number of processor cores. While both CCSP C and CCSP occam-$\pi$ implementations use the same runtime, the occam-$\pi$ compiler caches

17

Table 1: Communication times, calculated using process ring results.

| Implementation | 1-core (ns) | 8-core (ns) |
|---|---|---|
| CCSP C | 10 | 67 |
| CCSP occam-$\pi$ | 11 | 32 |
| Erlang | 428 | 404 |
| Haskell | 71 | 3448 |
| Java | 5579 | 5221 |
| pthread C | 2236 | 2535 |

scheduling pointers in registers, reducing the kernel call overhead. This explains the divergence in the results.

Haskell performance degrades with the addition of cores. We suspect this reflects internal contention exposed by multiple processors accessing the Haskell runtime in parallel.

The plot in Figure 13 shows the time taken for 1024 circulations of 64 concurrent tokens as the number of processor cores is increased. With the exception of POSIX threads and POSIX threads-based Java, all the implementations show decreased performance with increasing numbers of cores. This reflects the fact that, for user processes, communicating between processor cores is more expensive than simulated communication on the same core. As the number of concurrent processes increases, they are scheduled on to separate cores, increasing the communication costs. In particular, the NUMA aspects of the system beyond four cores show up as a significant degradation of performance.

POSIX threads and Java performance is noticeably improved by more cores. We speculate that threads are being given processor affinity by the Linux scheduler. This then improves performance as interprocessor communication via processor caches is faster than Linux's context-switch.

While Erlang performance is stable, Haskell performance notably degrades with increasing numbers of cores.

Our runtime, while not performing as in the optimal case (single-core), does control the slow down with increasing numbers of cores. We would not expect performance to degrade below interprocessor communication time.

*5.3. Agent Simulation*

As previously mentioned, occam-$\pi$ is being used for complex systems modelling as part of the CoSMoS project [34]. The investigators are explor-
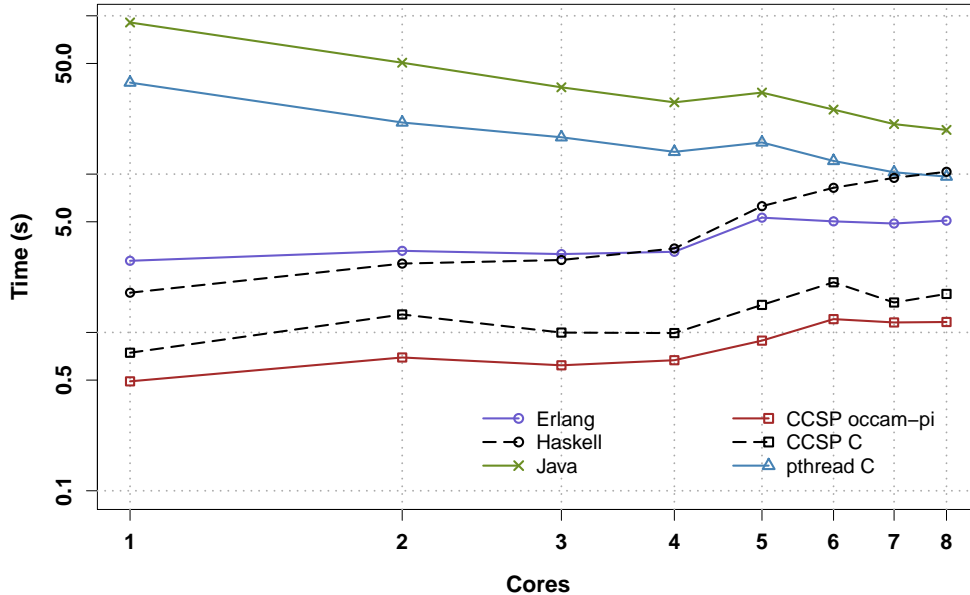
Figure 13: With 64 tokens in the process ring, we increase the number of processor cores.

ing using process-oriented methodologies for building models of emergent behaviour, and creating a generic toolkit for doing so. One of the early models investigated by the group was a process-oriented implementation of Craig Reynolds' *boids*, a simulation of flocking behaviour [35]. The CoSMoS project's implementation, *occoids*, employs *agent* processes with internal concurrency to implement the boids and their behaviour rules [19]. Agent processes move through a grid of *location* processes, connecting and reconnecting as they go. The topology of space can be modified by adjusting the underlying network connections, and this technique has been exploited to build an implementation which spans a network of computers with only minor changes to the code base.

We have constructed a benchmark based on occoids. Our benchmark is designed to be easy to implement in other languages, and produces results which allow the verification of an implementation's correctness. The simulated space is a two-dimensional torus, and agent positions are represented as integers relative to the centre of their present location. The occoids simulation uses floating-point variables so as not to unduly quantise space; however, integers allow us to easily verify the simulation output and avoid any associated variations in floating-point support.
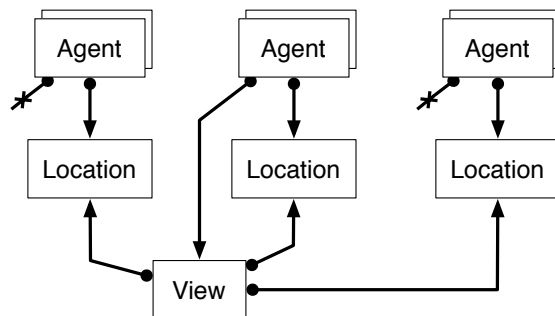
Figure 14: Simplified occoids process diagram. Boxes represent concurrent processes. Arrows represent two-way client-server channel connections, with the arrow pointing at the server. Agent processes connect to their present location, and "see" other agents via the location's view.

With reference to the process diagram in Figure 14: Location processes, acting as servers, maintain a data structure containing all agents presently in their grid area. View processes act as servers to clients, but also as clients to the location processes, building aggregate lists of all agents within nine adjacent locations each simulation step. Agent processes query a view process, and calculate a repulsive force from other visible agents, applying an internal *bias*. Having determined the force, the agent signals movement to its location, reconnecting to a new location if appropriate. Agents maintain a consistent sense of time using barrier synchronisations between activity phases.

The bias is updated based on the position of the agent and the number of other agents seen. In effect the bias produces randomised behaviour in the agents. The initial position of all other agents in the simulation acts as the seed, and hence can be easily reproduced.

As a comparison to the process-oriented design, we implemented a hand-optimised data parallel version using POSIX threads. Only one thread is used per processor core, and each thread executes a fixed number of agents. Data updates are performed in parallel using fine-grain locking of location data structures. This version represents the optimal case and appears as *pthread DP C* in Figures 15 and 16.

Figure 15 shows comparative results as we increase the number of available processor cores with a fixed-size world grid and number of agents. With reference to the process-oriented implementations, our runtime provides a marked improvement in performance and scalability. Haskell fails to achieve more than 50% speed up, even with eight available processors. POSIX
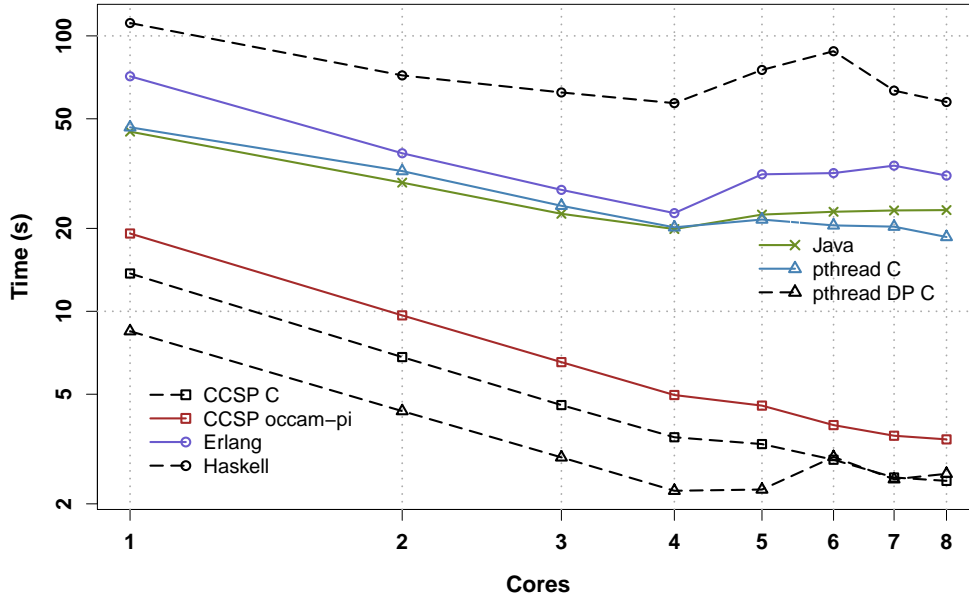
Figure 15: Increasing the number of cores applied to the agent simulation. The simulation is a 10x10 grid with 1200 agent processes.

threads achieves approximately a 150% speed up over eight cores, while our runtime achieves 500%. Scalability of our runtime also outstrips the ideal POSIX threads solution over eight cores.

Again, the NUMA nature of the architecture changes the performance profile of the machine after four cores. This negatively effects the performance of all implementations except those using our runtime.

The overall performance of the C version using our runtime is 50% of the optimal case. Assuming this performance loss is communication and scheduling overhead then further refinements of our scheduler and compiler integration should be able to bring performance closer to the optimal case. The reduced performance of occam-$\pi$ compared to C is due to more efficient optimisation of serial code by the GNU C compiler than the occam-$\pi$ compiler. We plan to overcome this by targeting GNU C or LLVM assembly as part of a new compiler we are developing [36].

Figure 16 shows results when scaling the simulation size with eight cores. Simulation size is controlled by increasing the grid size and number of agents. In this test our runtime also outperforms other process-oriented implementations. The other process-oriented implementations increasingly diverge
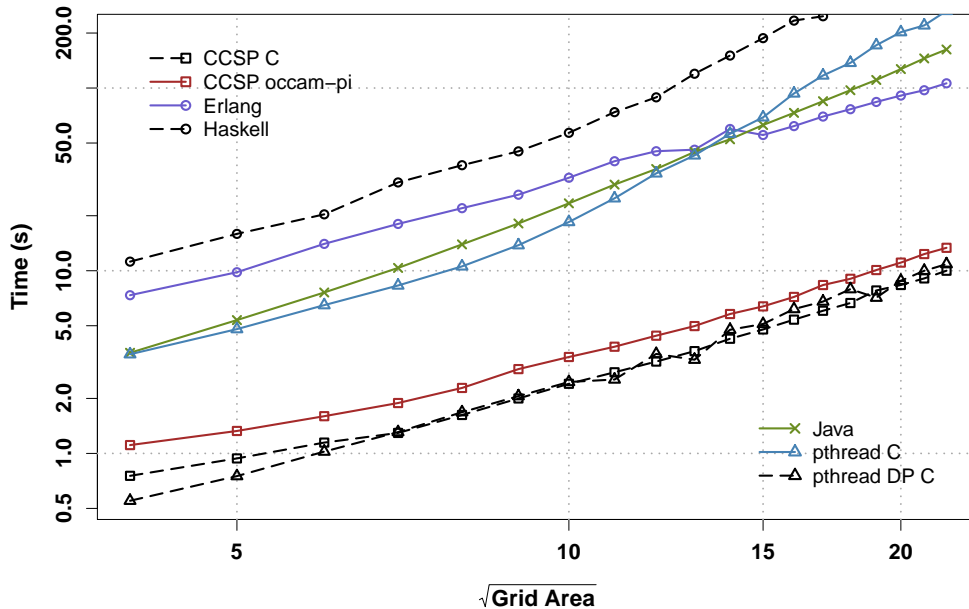
Figure 16: Simulation time for agents benchmark with increasing grid size. Each grid location has 12 initial agents. The x-axis is the number of locations in each axis.

from the optimal case with increasing problem size. The similarity of our runtime's scaling curve to the optimal suggests that refining of our existing runtime may be sufficient to achieve near optimal performance.

## 6. Related Work

Many frameworks and languages provide concurrency primitives beyond those supported by OS threads and locks. This stems from a desire to make concurrent programming easier, and to avoid common errors associated with locks and shared-memory [37]. For example message-passing frameworks such as PVM [38] and MPI [2] provide primitives similar to those presented in this paper, but do so for a course-grain network environment.

It is also desirable to provide lightweight concurrency primitives when the number of concurrent elements is high [39], or the application has more complete information on how they interact and should be scheduled. Of particular relevance to our work are lightweight runtimes for task parallelism such as Cilk [40], OpenMP [41] and Intel's Thread Building Blocks (TBB) [42]. These runtimes employ modern work-stealing scheduler designs

similar to our own, but do not provide primitives suitable for implementing process-oriented designs.

OpenMP and Intel's TBB emphasize the data parallelism of tasks, and only provide for communication of data asynchronously via shared memory. Neither framework provides constructs for communicating data with synchronisation. OpenMP's mutual exclusion locks can be used to implement communication channels. However, unlike POSIX threads, there is no conditional variable which can be used to efficiently implement resume on data or buffer space availability. While TBB's `concurrent_queue` provides a communication channel like interface, TBB only permits parallel tasks over ranges of data and does not support the spawning of continuously running tasks.

Programming environments such as Cilk [40] and Java's Fork/Join framework [43] focus on scheduling finite tasks with well-structured computational dependencies (directed acyclic graphs). Within these frameworks the dependency graph provides the scheduling scope and the depth of the graph can be used to bound the number of active tasks and memory utilisation. These bounding guarantees are based on the space requirements of the serial execution of the same program. Our process-oriented programs do not necessarily have a serial execution, so this model of space bounding is not applicable. Furthermore, as the lifetime of individual tasks is bounded, a LIFO scheduling order is appropriate. Lock-free operations on LIFO stacks are simpler than those on a FIFO queue. The processes we define in this paper have unbounded lifetimes and hence FIFO scheduling ensures all processes are serviced. A FIFO scheduling order distinguishes the scheduling algorithms presented in this paper from those of other work-stealing schedulers.

Process-oriented programming is very similar to the stream programming paradigm. Stream programs consist of graphs of concurrent communicating elements which transform input to output. Process-oriented programming is distinguished from stream programming in that it permits the dynamic creation of processes and their runtime reconnection, whereas a stream program's data graph is fixed which allows compile-time and instruction-level scheduling strategies [44].

In the benchmarks presented in this paper (section 5) we have focused on languages with clear support for implementing process-oriented designs, examining both Erlang and Haskell. Erlang provides asynchronous message passing, which can simulate communication channels, and has a shared-memory multiprocessor runtime [45]. Haskell as a pure functional language

focuses on deterministic parallel graph reduction rather than task interaction, but does provide a `MVar` primitive akin to a one-place buffered communication channel [33]. However, while both Erlang and Haskell provide support for lightweight concurrency, neither runtime (as tested), employs a work-stealing scheduler or lock-free algorithms for interprocess communication.

The Scala programming language provides an actor model of concurrency [46, 47]. This form of asynchronous message passing is similar to that provided by Erlang, except extended with asynchronous channels. However, where Erlang provides a single message reception primitive, Scala has two separate `receive` and `react` primitives. These correspond to thread-based operation (`receive`), and non-returning event-driven operation (`react`). The thread-based mode is underpinned by OS threads, whereas the event-driven mode is lightweight. Therefore the programmer must elect to use the lightweight construct and structure their program accordingly.

Concurrent ML (CML) is another functional language which provides lightweight concurrency primitives [48]. It implements channels and message passing using continuations on top of Standard ML. We excluded it from our comparisons as CML was not originally intended for multiprocessor execution. A successor language to CML, Manticore, attempts to address heterogeneous parallelism [49]. Manticore is still in the design and implementation phases and this prevented us making any performance comparisons.

Pict is process-oriented programming language based on the $\pi$-calculus. However, while the $\pi$-calculus and occam-$\pi$ use synchronous communication as a basic primitive, Pict favours asynchronous communication [50]. The language authors claim this decision was taken to improve performance and permit the implementation of certain language features. Specifically, the implementation of choice between communication channels in Pict relies on asynchronous communication as it is not a language primitive. This introduces the need for garbage collection semantics or a management process in order to prevent memory leaks [51]. The choice functionality detailed in this paper is a runtime primitive with constant memory overhead (section 4.1).

In summary, the runtime presented in this paper provides multicore scheduling for lightweight concurrent communicating processes which can be defined and reconnected at program run time. In doing so it provides support for process-oriented programming of multicore systems not provided by other frameworks for lightweight concurrency.

*6.1. RMoX*

An application and the principle funder of this work is the RMoX project [20]. RMoX is a process-oriented operating system, whose occam-$\pi$ processes are scheduled using the work described here. Use of the occam-$\pi$ language and the scheduler described provides a lightweight and scalable implementation, supporting large numbers of processes in the embedded systems for which RMoX is designed. Even though small (PC based) embedded systems are the primary target, RMoX is routinely tested on desktop systems and we expect it to scale successfully to future massively multicore systems, without heavy performance penalities, as the benchmarks here indicate.

## 7. Conclusions

We have implemented a multicore scheduler for fine-grain concurrent software developed using process-oriented programming. Process-oriented designs have a high degree of interprocess communication, and involve many more processes than physical processors. We address this in our runtime design by ensuring that:

- The serialisation bottleneck of a global run queue is avoided by scheduling processes independently on each core.

- Cache utilisation is improved by batching communicating processes.

- No programmer intervention is required to achieve multicore execution of process-oriented designs. Processes and batches are automatically distributed and migrated between processor cores.

- Contention within the scheduler is reduced using lock-free algorithms.

- Lock-free algorithm performance is optimised by minimising the number of atomic instructions, particularly in hot paths.

The performance results presented in this paper show that by addressing these points our runtime has significantly better performance than a number of other frameworks for implementing process-oriented designs. Specially, our runtime brings the performance of process-oriented software close to that of optimised multithreaded implementations.

Using the runtime presented in this paper, process-oriented design can be applied to develop software for multicore systems without the associated

complexities and hazards of threads, locks and shared-memory. Furthermore, we expect refinements of our runtime design to be able to allow unmodified process-oriented software to fully utilise hardware parallelism in future generations of multicore processors [11, 52–54].

## 8. Future Work

As presented, our runtime does not provide any asynchronous communication mechanism. Instead, we implement asynchronous messaging using buffer processes on synchronous channels. While this design decision was influenced by the target occam-$\pi$, a language with no asynchronous communication primitives, it may be that asynchronous communication warrants direct implementation. An investigation of the impact of asynchronous communication on the performance and expressibility of complex systems simulations is required. It should also be noted that there is an argument for synchronous channels being easier for developers to reason about and formally verify.

Further benchmark comparisons of our work are required to provide a comprehensive picture of performance. In particular, exploring the caveats of the heuristics detailed here. Research into process-oriented implementations of other common benchmark suites is part of our future work. One possibility is to reimplement benchmarks developed for stream programs, such as the StreamIt benchmark suite [44, 55] – although, as noted in section 6, these do not deal with the dynamic nature of process creation and communication in process-oriented programs.

As the number of processor cores increases, most multiprocessor architectures develop *non-uniform memory architecture* (NUMA) characteristics. To allow increased scalability to this class of machine, we need to add NUMA-awareness to our scheduler and memory management subsystems, and reassess the use of atomically mututable bitsets. One promising solution is to dynamically constrain subsets of the process network and their associated batches to groups of logical processors.

Finally, our present runtime contains many IA-32 architecture specific elements. We plan to enhance portability and experiment with possible compiler integration using LLVM, a technique being investigated for software transactional memory [56]. At the same time it is our aim to maintain a very small code base, so as to allow the targeting of embedded systems, such as those supported by other runtimes for occam-$\pi$ [57].

*References*

1. P. H. Welch, F. R. M. Barnes, Mobile Data Types for Communicating Processes, in: H.R.Arabnia (Ed.), Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications 2001, Vol. 1, CSREA Press, 2001, pp. 20–26.

2. W. Gropp, E. Lusk, R. Thakur, Using MPI: Portable Parallel Programming With the Message-Passing Interface, MIT Press, 1994.

3. A. C. Sodan, Message-passing and shared-data programming models - wish vs. reality, HPCS 2005 (2005) 131–139.

4. W. Athas, C. Seitz, Multicomputers: message-passing concurrent computers, Computer 21 (8) (1988) 9–24.

5. M. D. Hill, J. R. Larus, D. A. Wood, Tempest: a substrate for portable parallel programs, Compcon '95 (1995) 327–333.

6. D. May, OCCAM, ACM SIGPLAN Notices 18 (4) (1983) 69–79.

7. I. Foster, Compositional parallel programming languages, ACM Trans. Program. Lang. Syst. 18 (4) (1996) 454–476.

8. R. Milner, Communicating and Mobile Systems: the Pi-Calculus, Cambridge University Press, 1999.

9. P. H. Welch, F. R. M. Barnes, Communicating mobile processes: introducing occam-pi, in: A. Abdallah, C. Jones, J. Sanders (Eds.), 25 Years of CSP, Vol. 3525 of Lecture Notes in Computer Science, Springer Verlag, 2005, pp. 175–210.

10. A. W. Roscoe, C. A. R. Hoare, R. Bird, The Theory and Practice of Concurrency, Prentice Hall PTR, Upper Saddle River, NJ, USA, 2005.

11. K. Vella, Seamless Parallel Computing on Heterogeneous Networks of Multiprocessor Workstations, Ph.D. thesis, University of Kent (December 1998).

12. M. Schweigler, A Unified Model for Inter- and Intra-processor Concurrency, Ph.D. thesis, University of Kent, Canterbury, Kent, CT2 7NF, United Kingdom (August 2006).

13. K. Debattista, K. Vella, J. Cordina, Wait-free cache-affinity thread scheduling, IEE Proceedings Software 150 (2) (2003) 137–146.

14. R. D. Blumofe, C. E. Leiserson, Scheduling multithreaded computations by work stealing, J. ACM 46 (5) (1999) 720–748.

15. I. M. Barron, The transputer, in: MiniMicro West 83, San Francisco, CA, Vol. 2(5), 1983, pp. 1–8.

16. INMOS Limited, The T9000 Transputer Instruction Set Manual, SGS-Thompson Microelectronics, 1993, Document number: 72 TRN 240 01.

17. F. R. M. Barnes, Dynamics and Pragmatics for High Performance Concurrency, Ph.D. thesis, University of Kent (Jun. 2003).

18. C. A. R. Hoare, Communicating Sequential Processes, Communications of the ACM 21 (8) (1978) 666–677.

19. P. S. Andrews, A. T. Sampson, J. M. Bjørndalen, S. Stepney, J. Timmis, D. N. Warren, P. H. Welch, Investigating patterns for process-oriented modelling and simulation of space in complex systems, in: S. Bullock, J. Noble, R. A. Watson, M. A. Bedau (Eds.), Proceedings of the Eleventh International Conference on Artificial Life, MIT Press, Cambridge, MA, USA, 2008.

20. F. R. M. Barnes, A framework for lightweight, flexible and concurrent operating-systems, EPSRC grant EP/D061822/1. http://rmox.net/ (Feb. 2006).

21. C. G. Ritson, P. H. Welch, A process-oriented architecture for complex system modelling, in: A. A. McEwan, S. Schneider, W. Ifill, P. H. Welch (Eds.), Communicating Process Architectures 2007, Vol. 65 of Concurrent Systems Engineering Series, IOS Press, Amsterdam, The Netherlands, 2007, pp. 249–266.

22. G. M. Amdahl, Validity of the single processor approach to achieving large-scale computing capabilities, in: AFIPS Conference Proceedings, Vol. 30, 1967, pp. 483–485.

23. A. Kaieda, Y. Nakayama, A. Tanaka, T. Horikawa, T. Kurasugi, I. Kino, Analysis and measurement of the effect of kernel locks in SMP systems, Concurrency and Computation: Practice and Experience 13 (2) (2001) 141–152.

24. M. Herlihy, A methodology for implementing highly concurrent data objects, ACM Trans. Program. Lang. Syst. 15 (5) (1993) 745–770.

25. M. Herlihy, Wait-free synchronization, ACM Trans. Program. Lang. Syst. 13 (1) (1991) 124–149.

26. H. Attiya, N. Lynch, N. Shavit, Are wait-free algorithms fast?, J. ACM 41 (4) (1994) 725–763.

27. K. Fraser, T. Harris, Concurrent programming without locks, ACM Trans. Comput. Syst. 25 (2) (2007) 5.

28. D. L. Weaver, T. Germond (Eds.), The SPARC Architecture Manual: Version 9, Prentice-Hall, 1994.

29. P. H. Welch, G. R. Justo, C. J. Willcock, Higher-Level Paradigms for Deadlock-Free High-Performance Systems, in: R. Grebe, J. Hektor, S. Hilton, M. Jane, P. Welch (Eds.), Transputer Applications and Systems '93, Proceedings of the 1993 World Transputer Congress, Vol. 2, IOS Press, Aachen, Germany, 1993, pp. 981–1004.

30. P. H. Welch, F. R. M. Barnes, Mobile Barriers for occam-pi: Semantics, Implementation and Application, in: J. Broenink, H. Roebbers, J. Sunter, P. Welch, D. Wood (Eds.), Communicating Process Architectures 2005, Vol. 63 of Concurrent Systems Engineering Series, IOS Press, IOS Press, The Netherlands, 2005, pp. 289–316.

31. CCSP Comparison Benchmarks, http://projects.cs.kent.ac.uk/projects/kroc/trac/wiki/CCSP.

32. M. Pettersson, K. Sagonas, E. Johansson, The HiPE/x86 Erlang Compiler, Functional and Logic Programming 2441/2002 (2002) 228–244.

33. T. Harris, S. Marlow, S. Peyton-Jones, Haskell on a shared-memory multiprocessor, in: Haskell '05: ACM SIGPLAN workshop on Haskell, ACM, New York, NY, USA, 2005, pp. 49–61.

34. Complex Systems Modelling and Simulation infrastructure (CoSMoS), http://www.cosmos-research.org.

35. C. W. Reynolds, Flocks, herds and schools: A distributed behavioral model, in: SIGGRAPH '87: Proceedings of the 14th annual conference on Computer graphics and interactive techniques, ACM, New York, NY, USA, 1987, pp. 25–34.

36. C. G. Ritson, Translating ETC to LLVM Assembly, in: P. H. Welch (Ed.), Communicating Process Architectures 2009, Vol. 67 of Concurrent Systems Engineering Series, IOS Press, Amsterdam, The Netherlands, 2009, pp. 145–158.

37. S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, T. Anderson, Eraser: a dynamic data race detector for multithreaded programs, ACM Trans. Comput. Syst. 15 (4) (1997) 391–411.

38. A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, V. Sunderam, Parallel Virtual Machine: A Users' Guide and Tutorial for Networked Parallel Computing, MIT Press, 1994.

39. T. V. Eicken, D. E. Culler, S. C. Goldstein, K. E. Schauser, Active messages: a mechanism for integrated communication and computation, in: Proceedings of the 19th Annual International Symposium on Computer Architecture, 1992, pp. 256–266.

40. R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, Y. Zhou, Cilk: an efficient multithreaded runtime system, in: PPOPP '95: Proceedings of the fifth ACM SIGPLAN symposium on principles and practice of parallel programming, ACM, New York, NY, USA, 1995, pp. 207–216.

41. OpenMP Application Program Interface, Version 3.0 (May 2008).

42. Intel Threading Building Blocks 2.1, http://www.intel.com/software/products/tbb/.

43. D. Lea, A Java Fork/Join Framework, in: JAVA '00: Proceedings of the ACM 2000 conference on Java Grande, ACM, New York, NY, USA, 2000, pp. 36–43.

44. M. Kudlur, S. Mahlke, Orchestrating the execution of stream programs on multicore platforms, SIGPLAN Not. 43 (6) (2008) 114–124.

45. J. Armstrong, R. Virding, C. Wikström, M. Williams, Concurrent Programming in Erlang, 2nd Edition, Prentice-Hall, 1996.

46. C. Hewitt, Viewing control structures as patterns of passing messages, Artificial Intelligence 8 (1977) 323–364.

47. M. Odersky, M. Odersky, Scala actors: Unifying thread-based and event-based programming, Theoretical Computer Science.

48. J. H. Reppy, Concurrent programming in ML, Cambridge University Press, New York, NY, USA, 1999.

49. M. Fluet, M. Rainey, J. Reppy, A. Shaw, Y. Xiao, Manticore: a heterogeneous parallel language, in: DAMP '07, ACM, New York, NY, USA, 2007, pp. 37–44.

50. B. C. Pierce, D. N. Turner, Pict: A programming language based on the pi-calculus, in: Proof, Language and Interaction: Essays in Honour of Robin Milner, MIT Press, 1997, pp. 455–494.

51. B. C. Pierce, D. N. Turner, Concurrent objects in a process calculus, in: Theory and Practice of Parallel Programming, Springer-Verlag, 1995, pp. 187–215.

52. M. D. May, P. W. Thompson, P. H. Welch, Networks, Routers and Transputers, IOS Press, Amsterdam, 1993.

53. A. Acharya, M. Tambe, A. Gupta, Implementation of production systems on message-passing computers, IEEE Transactions on Parallel and Distributed Systems 3 (4) (1992) 477–487.

54. H. Lu, S. Dwarkadas, A. Cox, W. Zwaenepoel, Message passing versus distributed shared memory on networks of workstations, Supercomputing, 1995. Proceedings of the IEEE/ACM SC95 (1995) 37–37.

55. M. I. Gordon, W. Thies, S. Amarasinghe, Exploiting coarse-grained task, data, and pipeline parallelism in stream programs, SIGARCH Comput. Archit. News 34 (5) (2006) 151–162.

56. P. Felber, C. Fetzer, U. Müller, T. Riegel, M. Süßkraut, H. Sturzrehm, Trans-actifying applications using an open compiler framework, in: TRANSACT, 2007.

57. C. L. Jacobsen, M. C. Jadud, The Transterpreter: A Transputer Interpreter, in: I. R. East, D. Duce, M. Green, J. M. R. Martin, P. H. Welch (Eds.), Communicating Process Architectures 2004, Vol. 62 of Concurrent Systems Engineering Series, IOS Press, 2004, pp. 99–106.