Mobile Processes, Mobile Channels and Complex Dynamic Systems

Eric Bonnici and Peter H. Welch

Abstract— **This paper explores a process-oriented approach to complex systems design, using massive fine-grained concurrency, mobile channels and mobile processes. The complex systems studied are self-organising, with emergent and evolving behaviours (apparent at the global level) arising from massive interactions between relatively simple components (that have only local knowledge). Classical ant foraging is used as a case study. Processes are used to represent space, environmental factors and the ants themselves. Ant processes (like all processes) only have knowledge of their internal state (looking for food, looking for their nest) and what they can glean from their local neighbourhood (by interacting with the processes making up that neighbourhood). The networks constructed are dynamic, changing as the ants move around and environmental factors are introduced and modified. The paper reports on two mechanisms for achieving this: channel mobility and process mobility. The language for implementation is occam-π, which has the necessary concurrency mechanisms built in as fundamental primitives and whose semantics is rooted in the process algebras of CSP and the π-calculus. Performance figures are given, including speedup curves for multicores, and some conclusions drawn.**

I. Introduction

Complex systems are the result of emergent behaviour arising from the mass interaction of underlying components whose individual behaviours are relatively simple and where the rules of interaction are also simple [14], [3], [4]. Examples include social insects (such as bees, ants, and termites) which display interesting, unexpected and complex behaviour when working together. On its own, of course, an insect is still a very complex system. However, the complexities of individual insects are not sufficient to explain the complexity of social insect colonies [14], meaning that a group of insects can display behaviour that is far more complex than any of the individuals taking part. The complex cooperation needed for this arises without any need for guidance from a leader or the existence of a global plan [4]. No central control may have other advantages; for example, if part of the colony gets destroyed whilst foraging for food, the rest of the society can remain unaffected.

Our thesis is that programming models need to reflect these mechanisms *to keep them simple* and that process oriented concurrency is a good candidate for that reflection. occam-π is a language that provides such a programming model. It gives direct expression to concurrency, including processes, communication, barriers, choice (prioritised and non-deterministic), dynamic network construction, mobile channels, mobile barriers and mobile processes. Its overheads are sufficiently light to support millions of fine-grained processes on standard microprocessors; its runtime kernel automatically and efficiently exploits multicore (without programmer intervention) and it distributes over clusters of machines (with programmer intervention). Stable reasoning about their construction (e.g. the absence of deadlock and race hazards) follows from the formal and compositional process algebras (CSP and π-calculus) that underlie its definition. It has significant potential for modelling complex systems, including biological mechanisms [11], and we explore part of this in this paper.

Eric Bonnici and Peter H. Welch are with the Computing Laboratory, University of Kent, CT2 7NF, UK: email {eb708,p.h.welch}@kent.ac.uk.

II. Complex Dynamic Systems and occam-π

occam-π is an imperative state-full language built around the concurrency model of Hoare's CSP [5], [12]. Compiler enforced language rules prevent unsynchronised access to shared resources, so that no data race hazards can happen. Strict aliasing control enables this and provides a simple semantics for assignment.

occam-π extends the classical occam2.1 language [6] through the careful blending in of dynamic mechanisms from Milner's π-calculus [8] – in particular, mobile channels, barriers and processes [17], [18], [11], [16], [2].

A full summary is not possible in this paper and the reader is referred to the last group of references cited above and the extensive on-line documentation – such as [10]. We limit ourselves here to the key ideas relevant to this paper.

A. Processes

A process encapsulates state and runs in its own thread, or threads, of control. Unlike an *object*, which relies on external threads to execute its methods, a process is in charge of its own behaviour (which makes them easier to design, program and reason about). Processes interact through synchronising on *shared events* (e.g. communication channels and multiway barriers). A *network* of synchronising processes is itself a process, just at a higher level of abstraction. A process may refuse any event (e.g. *take-this-stuff*) at any time for internal reasons (e.g. *no-room*). This constrains the behaviour of the network in which it is embedded, shaping that for safe and correct higher-level function. Communication and synchronisation also enable safe access to data. With the additional checks mandated by occam-π, the complete absence of race hazards is guaranteed. Classical skills of serial programming remain valid – no locking algorithms are needed to ensure that concurrency introduces no surprises. The semantics of concurrency are *compositional*.

B. Channel Bundles, Servers and Clients

Processes communicate through zero-buffered strongly typed channels. A sending process blocks until its message is taken. A receiving process blocks until a message is

978-1-4244-2959-2/09/$25.00 © 2009 IEEE

sent. When needed, buffered channels are easy to introduce with extra processes. Communication occurs between a single sender and receiver; multiple senders (respectively receivers) may *share* the same channel-end but only one sender (receiver) at a time may use it, with competing senders (receivers) queued. Channels may be grouped into *bundles*, with each element carrying a different message structure and used in differing directions.

Client-server systems typically consist of *server* processes waiting for a request on one element within a channel bundle, the end of which is exclusive to that server, and returning information on another channel element. *Client* processes make requests and receive answers from the other end of the channel bundle, which is shared by all clients. A client-server transaction is *atomic* – it cannot be interleaved with transactions with other clients on the *same* bundle.

Process networks built purely from client-server relationships and which have no cycles in those relationships are *deadlock free* [7]. The models outlined in this paper make heavy use of client-server relations (with regions of space modelled by servers and passing traffic modelled by clients), but they are not pure client-server systems.

C. Barriers and Phases

Barriers, [18], enable many processes to synchronise. When one process offers to synchronise on a barrier, it blocks until *all* processes registered on that barrier also synchronise – the last one to do so releases all the others. Barrier registration is automatic and cannot be side-stepped – if a process has a reference to a barrier, it is registered. Processes may be registered on many barriers. The registration set of a barrier is dynamic as processes acquire reference, lose reference or terminate.

In our models, client processes (e.g. ants) use barriers to coordinate access to the space servers into time-distinct *phases* [18], [11]. In each cycle (i.e. time step of the simulation), each client goes through two phases – each one scheduled by a barrier:

```
while alive
  seq
    sync observe
    ... observe the world (interact with local sites)
    sync modify
    ... change the world (interact with local sites)
```

The above occam-π *pseudo-code* shows the main loop structure for these clients. The *observe* barrier ensures that all processes enter their *observe the world* phases together: they all see the same world view, as nothing changes during this period. Having made their plan(s), they wait for each other on the *modify* barrier before entering their *change the world* phases together: in this period, they may be competing with each other for limited resources (e.g. to move to the same site), changing their local environment (e.g. depositing pheromone) and actually moving themselves. Competition is resolved by the site servers.

Within individual phases, the whole system behaves as a pure client-server network, with no client-server cycles, and is therefore deadlock free. Thus, each phase terminates. We may now ignore those phases and consider the remaining behaviour. This comprises all the client processes looping through the above cycles, with new ones appearing and old one terminating at arbitrary moments. However, the occam-π rules (enforced at compile time) ensure that all alive processes are registered on the two barriers and, since they all synchronise on them in the same sequence and in every cycle, the system is deadlock free at this level. Therefore, the whole system is deadlock free.

The occam-π language rules also ensure (at compile time) that no race conditions on shared data is possible. So we can be confident of these basic safety elements in our design.

D. Dynamics and Mobility

Both channels and processes may be constructed at runtime. Network topologies, even between existing processes, may change.

There are two ways of doing this: channel mobility and process mobility. In a sense that we are still discovering, these are *dual* notions. Network dynamics can be captured by either, but the ease and efficiency of capture varies. In some circumstances, mobile channels work better; in others, we should use mobile processes. In work reported to date (e.g. [11]), only mobile channels have been considered. Apart from acceptance and performance trials, this is the first exploration (of which we are aware) of the mobile process mechanisms within occam-π.

In addition to traditional data, channels may carry *ends* of channels (actually ends of channel bundles). For example, a client process may construct a new channel bundle, send one end down a channel to a server and keep the other. The server may fork a temporary process to service the client through the received bundle, in parallel with other client transactions. Alternatively, the server may forward that channel bundle end to another server that is better equipped to deal with it. Either way, the original client has a new connection to a process with which it was previously unconnected.

An ant process holds the client-end of a channel bundle to its current site. It can request the client-end of a bundle to a neighbouring site (always held by the site) to be sent over the reply channel in the bundle currently held. Once received, it can let go its previous channel bundle and it is connected to its new site – the ant has *moved*.

Channels may also carry *processes*. These must be specially declared as mobile process types having a specific interface of synchronisation parameters (which we can think of as sockets into which channels must be plugged before it can be activated). Instances are referenced through variables and constructed at runtime. Initially passive, they may be moved through channels like any ordinary data. Language rules ensure that only one process at a time has reference to any individual mobile element – communication of a mobile really *moves* it (i.e. the sending process loses it) [1].

A process holding a mobile process (its *platform*) may activate it by plugging it into a set of locally available and compatible channels. The mobile process then runs and can

use the channels provided. At any time, the mobile process may *suspend* its activity and become passive again. Control then returns to its platform process (which was suspended while its mobile runs), which may send it on through any channel that is typed to carry it. When re-activated by some other platform, the mobile resumes from the exact point it suspended with all its private state intact, but now with the *different* set of external channels it has been given.

More details of how these different mechanisms are used in our ant models, together with performance comparisons, are given in the next section.

III. Ant Foraging Model

A process oriented model of ants foraging for food has been constructed. The architecture is massively parallel and fine-grained, modelling directly notions of space, locality and time. It also directly reflects the dynamics of a changing environment, the movement of autonomous agents within space and all interactions between all parties (fixed or mobile).

A. Architecture

This architecture evolves that described in [11], with adaptations to use mobile processes and allow many agents per *site*. The world is made up of many site processes connected through channels to form a two dimensional space. These site processes act as server processes responding to requests made by client processes. The client processes are ants (which move from site to site) and pheromones (which the ants deposit at sites, which do not move and which gradually evaporate). The site processes also do not move – unless we want to model a universe where the space topology is dynamic (e.g. for the introduction of *worm holes*!). Each site represents a different and limited area of space, allowing a limited number of ants to be present at any moment. No limit is imposed on the amount of pheromone present at a site (their molecules are much smaller than ants).

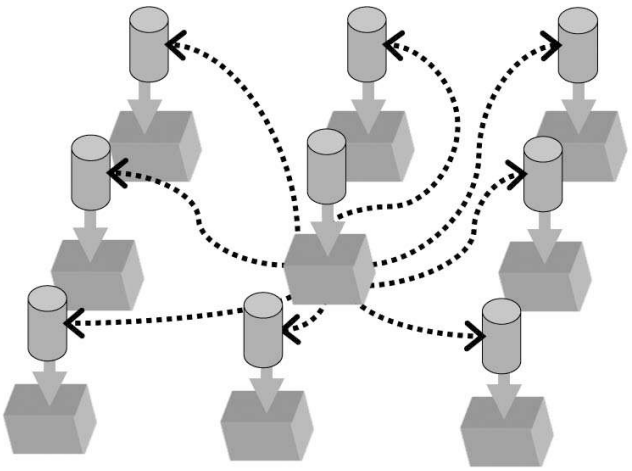

Fig. 1. The server processes

Each site services a channel bundle through which any number of clients can interact. Each site also has reference to the client-ends of its eight neighbour sites – not to use for communication with its neighbours (which would cause deadlock through client-server cycles), but to provide links that the ants may use to explore its locality and, if so minded, to move.

In Figure 1, the cubes represent the site processes, and the cylinders pointing down to them are the channel bundles which each site services. The dotted arrows pointing to neighbouring sites' channel bundles indicate references to the client ends of those bundles.

The ants have two states representing goals: looking for food and looking for the nest. Whenever a goal is reached, the ant is given a reward which allows it to deposit more pheromone making its trail stronger.

Pheromones are processes whose on-the-fly construction is triggered by ants arriving at virgin sites. They become attached to those sites. These processes control a level of pheromone (passive data) recorded in their sites, gradually reducing that strength over time and modelling *evaporation*. They are topped up by any further arrival of ants and terminate when (if) their controlled level reaches zero. In our current model, they do not replicate and diffuse to neighbouring sites, though they easily could.

Time is modelled by barrier synchronisation of the processes that must be aware of time: in this study, the ants and pheromones.

Site processes do not need to be aware of the passing of time. If there are no ants or pheromone present, they wait passively for something to appear. This waiting requires zero processor cycles: sites perform no active polling. A waiting site will be awakened (i.e. re-scheduled) by its next visitor. This saves much computation since there are a great many sites, most of which will be passive at any one time.

Visualisation of the evolving system is managed by mapping the state of each site (the number of ants present, the levels of pheromone, the presence of food, the presence of an obstacle) to a colour on a single pixel of a two dimensional graphics window. Each site shares its pixel with a rendering process that sees the whole pixel matrix. This sharing is made safe through the renderer synchronising with the ant and pheromone processes on the `observe` and `modify` barriers (section II-C) and only rendering in the *observe* phase (when nothing is changing). Users may interact with this visualisation (through mouse clicks) to introduce food and place obstacles. In this model, food and obstacles have no autonomous behaviour and are modelled by passive data (part of the state of the containing site).

B. Ants and Pheromone

When searching (i.e. not following a trail of pheromone), ants do not move completely at random. They move by a combination of random walk and a preference for moving forward, a preference that grows weaker each time it is satisfied. The greatest percentage chance for movement is given to the previous direction, with lower and lower percentages given to steeper and steeper turns. This prevents 180 degree

turns and results in more ant-like searching, rather than the Brownian motion effect of a pure random walk.

Other rules have been introduced. If an ant has not found food after searching for a defined number of steps, it gives up and tries to make its way back to the nest. However, a small probability is set for giving up the search for food before that defined maximum has been reached. This introduces further emergent overall behaviour, reducing the number of ants that get lost (i.e. cannot find a trail back to the nest).

Following a pheromone trail is not randomised: ants move towards the highest concentration, subject to there being room. However, another probability controls whether an ant following a pheromone trail abandons it and heads off at random. The benefits of this will be discussed in section IV.

Direction preference probabilities (and the rules for their decay), the *maximum search length*, and the *give up search anyway* and *abandon trail* probabilities were initially set by guesswork. They evolved to the set of values used by observing the resulting behaviours in the model and comparison with moving images of real ants. Of course, this would be an interesting challenge for an evolutionary algorithm – but we confess to doing this by hand (and eye).

Ants deposit pheromone on the site in which they currently reside by sending it a message containing the type and amount of pheromone to deposit. The site reacts by forking off a pheromone process to control its evaporation (unless one already exists for that particular variety). In our model, sites have no sense of time so cannot do this unaided. The pheromone processes, on the other hand, synchronise on the barriers and can track time.

Ants deposit two types of pheromone: *food* and *nest*. If an ant finds sites containing pheromone, it will *tend* to move to the neighbouring site containing the largest amount of pheromone relevant to its current goal (*find food* or *find its nest*), depositing the opposite pheromone just before it moves (reinforcing a trail back to the nest or the food).

70	70	70	70	70	70	70
70	80	80	80	80	80	70
70	80	90	90	90	80	70
70	80	90	100	90	80	70
70	80	90	90	90	80	70
70	80	80	80	80	80	70
70	70	70	70	70	70	70

Fig. 2. A layout of pheromone close to a nest

Figure 2 shows an example of pheromone intensity levels across a grid of site processes. In this case, the middle site is the nest and contains the greatest level. Levels decrease every step taken away from it. If an ant is following the nest pheromone back to the nest, then all it has to do is always travel towards the largest pheromone level that it can see from its current location. If there are equally large levels, then the ant chooses the one closest to its previous direction of travel (which means the ant prefers to keep moving in a straight line).

The simulation starts with all ants in the nest. As they move out, they search for food. Once a food source is discovered, the ant looks for home. Gradually, and just from the low-level rules programmed in the ant processes, a trail emerges as more and more ants discover the relevant pheromone signals. This trail formation is not explicitly programmed into the model but emerges from the mass interactions of sites, ants and pheromones. The trail gets stronger and straighter as it gets reinforced by continual use.

C. Mobility

Mobility in the system is achieved via one of two methods: either by using mobile channels or by using mobile processes.

1) Mobile Channels: Ants move between sites by asking its current site for the client end of the channel bundle serviced by its destination site. The current site has this reference, since the destination site will be one of its neighbours. The site delivers that channel bundle client end through one of the channels in its own bundle to the requesting ant. The requesting ant receives this new channel bundle end and lets go the one it was previously holding. It is now attached to the new site – it has moved.

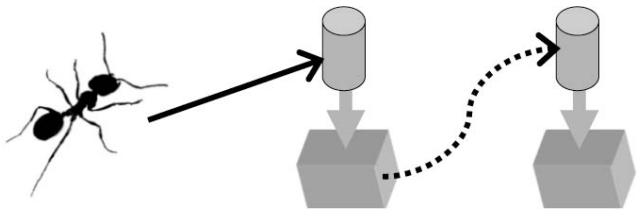

Fig. 3. Mobile channels (before ant movement)

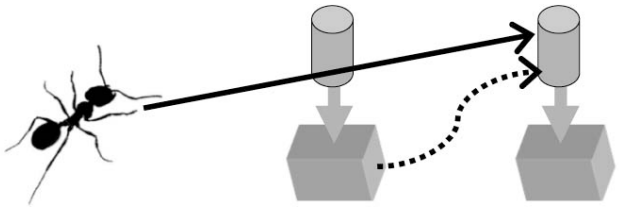

Fig. 4. Mobile channels (after ant movement)

Figure 3 shows an ant process communicating with the leftmost site process. This ant wishes to move to the process to the right. The ant requests and receives the client end of the destination site's channel bundle, attaches to it, and releases its grasp on the previous channel end – Figure 4. The ant has moved from one site process to another.

2) Mobile Processes: Instead of achieving mobility via the movement of channel ends, processes may be declared to be mobile themselves. Mobile processes must be plugged in (through channels) to an environment (other processes). To do this, they rely on a *platform* process which receives them, plugs them in and activates them and, when they suspend, sends them on their way again. A mobile process must suspend before it can move. Once suspended, the mobile process may be sent down a channel and reconnected within a different environment.

For some applications, the platforms that handle mobile processes can be permanent and pre-assigned. For our model, this platform cannot be the site process itself since, otherwise, it would not be able to to do anything once it had activated the mobile – it would have to await its suspension. So instead, the site *forks* off a special platform process just to manage the newly arrived mobile for the duration of its visit. This allows any number of mobiles to be plugged into the site (though the site limits that for its own reasons) and for the site to continue to service them. Fortunately, the overheads for forking processes, shutting them down and recovering their resources (e.g. memory) are extremely small in occam-π (section IV-B).

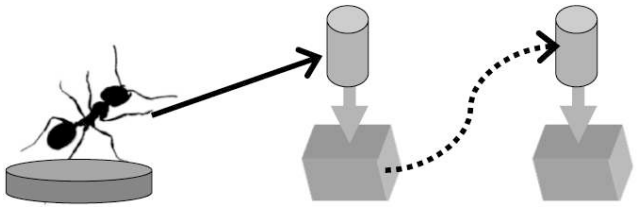

Fig. 5. Mobile processes (before ant movement)

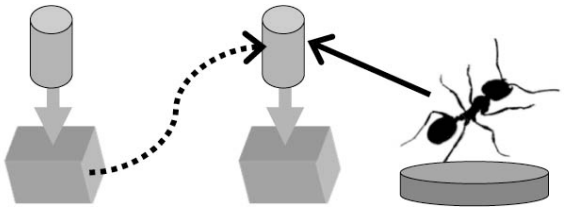

Fig. 6. Mobile processes (after ant movement)

In Figure 5, the ant process resides on a platform process – indicated by the disc under the picture of the ant. As in the previous example, the ant wishes to move to the right. From its current site, it gets the client end of the channel bundle serviced by its chosen destination, communicates that to its platform and suspends. The platform sends the ant process *through* one of the bundle channels to the destination site and terminates – its job is done. The destination site (which is already committed to accepting the new ant) receives it, forks off a new platform process and gives it the ant. The new platform plugs the ant into the new site – Figure 6. The ant has moved.

Each mobile process needs to be initialised upon creation. This is done via an initialisation channel, which is then never used after initialisation! When the process is moved and reconnected in a new environment, an *initialisation* channel still needs to be plugged in, even though it will never be used. Currently, this is handled by plugging in a dummy channel – which is trivial, but tedious and needlessly distracting (see section V).

IV. Observations and Performance

A. Emergent Behaviour

There should be no difference in the observed behaviour between the version using mobile channels and that using mobile processes. None was observed.

The simulation starts with all ants in the nest. All the ants start off looking for food. Initially, there is no pheromone anywhere in the simulated world and the ants move around at 'random' (see section III-B). The ants are looking for food, or food pheromone, which means that they deposit nest pheromone as they move. This allows them to find their way back to the nest once a food source is found. Once an ant locates a food source, it starts to head back home whilst depositing food pheromone to allow other ants (including itself) find the way to the food source later. Other ants in the vicinity of the food pheromone are able to detect this and follow it to the food source. Just from these low-level rules, a trail emerges as more and more ants discover the relevant pheromone signals. The trail gets stronger and straighter as it gets reinforced by continual use.

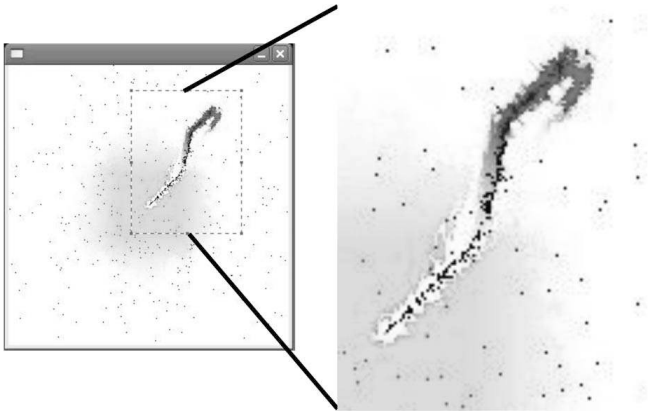

Fig. 7. A simple trail

Figure 7 shows such a trail of ants from a food source to the nest, which is gradually straightening. The nest is located in the middle of the screen with the food source located at the end of the trail in the upper right hand corner. The darkest spots are the ants. The different shades along the trail show food and nest pheromone intensity levels – the darker the shade, the more pheromone is present.

Once a trail is formed, it gets reinforced by other ants using it. Simply following a trail strengthens the intensity of pheromone laid upon it. Ants following *shorter* trails take less time than those following *longer* ones. Those ants, therefore, make more frequent trips, resulting in greater deposits of pheromone. So, shorter trails mean stronger trails.

This positive feedback recruits further ants, which prefer to join a stronger trail when given a choice (and making it stronger still). Since shortest paths are straight lines, trails that survive will tend to straight lines.

Longer trails that may have formed are eventually abandoned by the ants. These trails disappear over time as their pheromone evaporates. This evaporation plays a negative feedback role in the simulation, removing longer trails as their use dwindles. Such behaviours are not explicitly programmed into the simulation at any level, but are emergent properties arising from the mass interaction between the low-level components (ants, pheromones, sites) in the system.

However, if obstacles are present through which the ants may not pass, there may be no direct straight line between a nest and a food source.

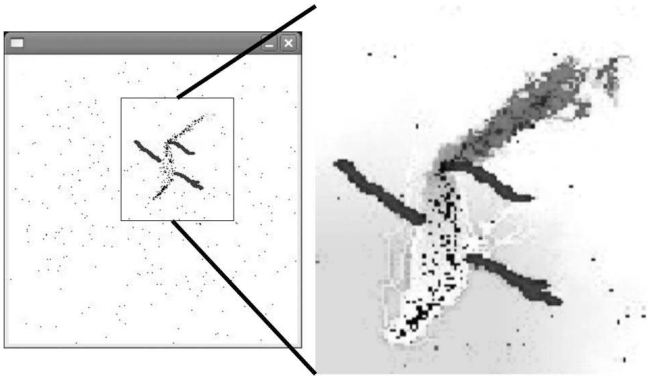

Fig. 8. An obstructed trail

Figure 8 shows some obstacles placed between the nest and food source (and over the trail shown in Figure 7). These obstacles are represented by the three parallel bands at right-angles to the original trail – someone has dropped three bricks! Sites containing obstacles refuse entry to ants, which must therefore choose some other path. Some ants may find one way round the obstacle and other ants the other. Those that find these ways lay down pheromone that attracts more ants and form new trails. But the path that survives will be the one that takes the shortest time to traverse – i.e. the shortest way round the obstacle is found. This classic route finding (e.g. [9]) is demonstrated in our models and shows further unprogrammed emergent behaviour. Of course, if after this route has become established the obstacles are removed, the trail gradually reverts to its original (single) straight line.

As mentioned in section III-B, an ant may decide spontaneously to abandon a trail it is following. This enables some of the ants to search out better paths than the one they are currently on, keeping the search space open. It also helps for dynamic environments in which obstacles are introduced or removed on the fly: the current best path may disappear (a brick is dropped) or cease to be the best path (a brick is picked up).

In the case of several food sources, if a closer one is added after a trail has been set up, then the ants still have a chance of finding it. Changing the value of this exploration factor results in a different thickness for the ant trail. If the exploration factor is set to a high value, then the chance of an ant abandoning a trail is high and the observed trail becomes thick. Decreasing the exploration factor results in a thinner trail being observed. The reason is that there are less ants abandoning and rejoining the trail than in the former case. The time it takes for a trail to settle into straight line(s) also changes. A thick trail settles more quickly than a thin one.

B. Benchmark Results

The mobile channel and mobile process versions of the ant foraging simulation were compared for their performance. The models are normally run with graphics visualisation rendering the state of the system every cycle. Since we are concerned with the impact of the different overheads for mobile channels versus mobile processes, the visualisation was set to render only once every 100 cycles. The models report counts of cycles-per-second every second.

No food sources were provided so as to promote stable repeatable results. No pheromone is deposited – its control process is not mobile, so its introduction would merely add noise to the overheads we want to compare.

Model behaviour follows the same pattern every run: the ants just spread out foraging.

Observing the cycles-per-second counts, the highest counts are at the start when all the ants are close to their nest. These counts descend to about two thirds of their original values as the ants spread out, bottoming out to a *"steady"* value within a minute (wall clock time). The steadiness of these values are modulo a noise level of plus-or-minus 1 (i.e. around 7%), almost certainly caused by noise in the Unix environment in which they were measured. The counts reported in Figure 9 are mean values, rounded to whole numbers, taken after 80 seconds (i.e. within the steady region). A detailed statistical analysis has not been conducted.

The reason for the decay in these counts is down to processor memory cache. At the start, only a few sites are being interrogated by all the ants. As they spread out, the number of sites holding ants approaches the number of ants (since we have *many* more sites than ants!). A greater number of active sites means more different memory requests each cycle and the number of cache misses will grow. Once the ants have spread out sufficiently so that there is mostly only one ant per site, more memory requests will be from cache lines not recently visited and, hence, lost. [This conclusion was confirmed by running the system with all ants initially dispersed randomly throughout the space: the lower *steady* cycle rates showed up immediately.]

Of course, if we placed food and watched trails emerge, the ants would become concentrated in the trails, memory requests would be similarly concentrated and the cycle rate should increase. But this is difficult to control in a repeatable way so does not play a part in our benchmark. For comparison of the effect of the overheads of mobile channels versus mobile processes, the *no-food* scenario suffices.

The parameters for our benchmark are as follows: 5000 ants on a grid of 600 by 600 sites. That is 360,000 site

processes, 5000 ant processes and a handful of others supporting visualisation, user interaction and other controls. The total number of active processes, at any particular moment, will be around 10,000 (one site per ant) once the system has reached the settled state at which the cycle count is taken. Note the site processes in this active set will be continually changing as the foraging ants move around.

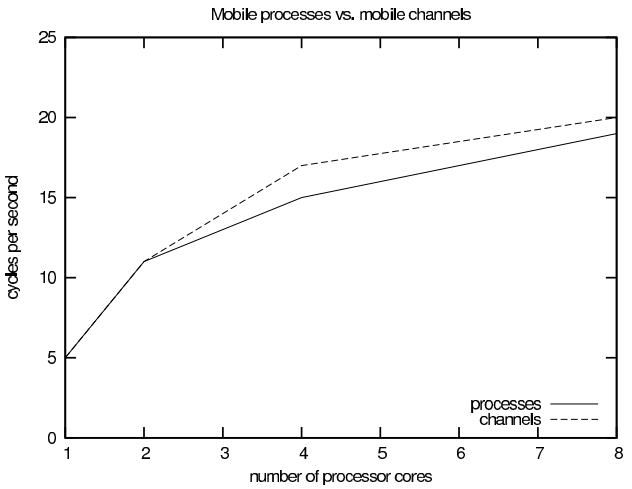

Fig. 9. Performance versus number of cores (ants dispersed and foraging)

The machine on which the benchmarks were run is an eight core Intel Xeon workstation comprising two E5320 quad-core processors running at 1.86GHz. Pairs of cores share 4MiB of L2 cache, giving a total of 16MiB L2 cache across eight cores. The benchmarks were run by forcing the code to restrict itself to using one, two, four or all eight cores (which is done by setting environment variables observed by the runtime - i.e. with no change in source or compiled code).

Figure 9 shows speedup curves for performance against the number of cores used. The two curves show the results for the mobile channel version (dashed line) and the mobile processes version (continuous line).

There are two immediate observations. Firstly, there is only a very modest performance penalty, for this application, from using mobile processes over mobile channels.

This is despite the higher actual overheads of implementing agent movement using mobile processes – especially with dynamically forked platforms (as in this system). To move a mobile process, it must first suspend, its platform must send it through the channel to its destination and then terminate, that destination must fork a new platform and give it the mobile and, finally, the new platform must re-activate the mobile. On the other hand, to move by receiving the channel bundle end of its next destination, all a process has to do is ask for it, receive it and discard its old connection.

Stress testing benchmarks ([17]) show that movement via mobile channels, with similar cache misses, costs around 100 nanoseconds. For movement via mobile processes, this cost quadruples. Actual costs depend, of course, on conditions in the running application – but these figures are in the right *ball park*.

On two cores, for example, there are approximately 11 cycles per second. That means 55,000 interrogations (per second) by the ants of their site processes leading to around 55,000 movements (per second), since the ants have spread out and will always be able to move. This means around one movement every 18 microseconds. The above cost estimates for ant movement mean, therefore, that they account for less than 3% of the whole load – even if mobile *processes* are used.

So, the results are not surprising but, nevertheless, good to see. They give us confidence that results will be similar for any application with a modest amount of work for agents to do in between movements.

The second observation is that real speed-up is obtained from multicore processors using the massively fine-grained concurrency technology built into the occam-π language, its compiler and runtime. For these models, speedup is super-linear from one to two cores, not bad from two to four, but drops off between four and eight.

Of course, normal occam-π applications would simply use as many cores as are available. No source or compiled code changes are needed to take advantage of this. It just happens.

V. Conclusions

Classical emergent behaviours from an ant colony foraging for food have been reproduced using a modelling and simulation approach that is *non-classical*, but which closely reflects *real-world* mechanisms. System design and implementation is massively parallel, with any item exhibiting non-trivial autonomous behaviour (e.g. a hungry ant, an evaporating pheromone, a location in space) represented by a *process*.

The networks built and reported here for the ant foraging model are *dynamic*, with the number of processes, the number of connecting events and their topologies under continual change. These dynamics closely follow similar dynamics in the real world. There is no particular difficulty in this approach – indeed, an important point is its simplicity, which comes from its directness.

This paper has focussed on two alternative mechanisms for achieving these dynamics: mobile channels and mobile processes. These are elements of the occam-π multiprocessing language, whose language rules, compiler and runtime provide ultra-light realisation of all the concurrency mechanisms described. In previous work (e.g. [11]), only mobile channels have been used for the dynamics. We needed reassurance that mobile processes were actually practical and efficient for use in real applications (as opposed to acceptance trials).

Mobile channels and processes are *dual* mechanisms for mobility. We note that the former are simpler to program and have (roughly) half the overheads of the latter – although the overheads for both are very small. The latter is, for some, a more attractive mechanism since it reflects more directly the idea of movement.

For a system running on platforms with distributed memory (such as a workstation cluster), mobile processes would be efficient by default, since they really do move to the machine holding their next environment.

With mobile channels, the processes stay in their original machine and plug into the environment to which they have 'moved' over networked channels (and suffer much higher latencies and lower bandwidths for interaction).

This problem may be overcome through extra components added to the system design. When a mobile process needs to move between machines, it sends its state as raw data to a receiving process on the target machine and terminates. The receiving process forks off a new mobile process and sets its state with the received state. The process has *teleported*, leaving a dead body behind (which automatically recycles its memory just before dying – i.e. there is no need for garbage collection). Details are reported in another submission to CEC 2009 by other authors [13].

We must also note that, *at present*, the occam-π run-time does not support the communication of mobile processes between machines. Until that is implemented, a distributed system using mobile processes has to use the same fix described for mobile channels, when crossing memory boundaries.

There are, however, some problems with the mobile process model currently presented by occam-π. They are certainly somewhat trickier to program than mobile channels. One aspect of this is the single interface that mobile processes offer to their environments. Sometimes, a mobile process needs to offer different faces to the world: for example, when being initialised, used and decommissioned. Currently, its single interface must be the union of all those required and dummy channels must be plugged into those parts that a particular environment does not need. This is dangerous since care must be taken by the mobile not to use any channels holding dummies. This is a language design issue that this work has brought into focus.

The benchmark results show the practicality of the process oriented mechanisms reported and, especially, the practicality of occam-π. With liberal use of concurrency (over 10,000 processes active at any time from a pool of more than 370,000), the ants system still runs at around 11 cycles per second on a standard dual-core processor. Even when visualisation is turned on (i.e. images rendered every cycle), the count drops by only one or two. This is fast enough for rapid feedback from and interaction with such models.

The other information from the benchmarks is how well this massively concurrent fine-grained process oriented approach scales over multicores, thanks in large part to the efficiency of the underlying occam-π technology. Such results should not cause surprise. Having a sufficient surplus of logical concurrency over physical concurrency – the principle of *parallel slackness* – was one of the tenets of Valiant's "bridging model" for general purpose parallel computation, back in 1990 [15].

ACKNOWLEDGEMENTS

This work was partly supported by the CoSMoS project (`www.cosmos-research.org`, EPSRC grants EP/E053505/1 and EP/E049419/1). We are indebted to all our colleagues in this project and in the Concurrency subgroup at Kent for their insights, motivation, encouragement and technical skills. We are especially grateful to Fred Barnes, Adam Sampson and Carl Ritson for their work on the occam-π technology and to Carl for running the reported benchmarks on the dual quad-core processor at very short notice.

REFERENCES

[1] F.R.M. Barnes and P.H. Welch. Mobile Data Types for Communicating Processes. In *Proceedings of the 2001 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'2001)*, volume 1, pages 20–26. CSREA press, June 2001. ISBN: 1-892512-66-1.

[2] F.R.M. Barnes and P.H. Welch. Prioritised dynamic communicating and mobile processes. *IEE Proceedings – Software*, 150(2):121–136, April 2003.

[3] E. Bonabeau, M. Dorigo, and G. Theraulaz. *Swarm Intelligence From Natural to Artificial Systems*. Santa Fe Institute, Oxford University Press, New York, 1999.

[4] G. Flake. *The Computational Beauty of Nature: Computer Explorations of Fractals, Chaos, Complex Systems, and Adaptation*. The MIT Press, Cambridge, USA, 1998.

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

[6] Inmos Limited. *occam2 Reference Manual*. Prentice Hall, 1988. ISBN: 0-13-629312-3.

[7] J.M.R. Martin and P.H.Welch. A Design Strategy for Deadlock-free Concurrent Systems. *Transputer Communications*, 3(4):215–232, October 1996.

[8] R. Milner. *Communicating and Mobile Systems: the Pi-Calculus*. Cambridge University Press, 1999. ISBN-10: 0521658691, ISBN-13: 9780521658690.

[9] L. Panait and S. Luke. A Pheromone-Based Utility Model for Collaborative Foraging. In *Proceedings of the Third International Conference on Autonomous Agents and Multiagent Systems*, IEEE Conference Proceedings, pages 36–43. IEEE Vomputer Society, July 2004. ISBN: 1-58113-864-4.

[10] P.H.Welch. An occam-pi Quick Reference, 2008. `https://www.cs.kent.ac.uk/research/groups/sys/wiki/OccamPiReference`.

[11] Carl G. Ritson and Peter H. Welch. A Process-Oriented Architecture for Complex System Modelling. In Alistair A. McEwan, Steve Schneider, Wilson Ifill, and Peter Welch, editors, *Communicating Process Architectures 2007*, volume 65 of *Concurrent Systems Engineering Series*, pages 249–266, Amsterdam, The Netherlands, July 2007. IOS Press. ISBN: 978-1-58603-767-3.

[12] A.W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall, 1997. ISBN: 0-13-674409-5.

[13] A.T. Sampson, J.M. Bjørndalen, and P.S. Andrews. Birds on the Wall: Distributing a Process-Oriented Simulation. In A.M. Tyrell et al., editor, *Proceedings of the 2009 IEEE Congress on Evolutionary Computation*, IEEE Conference Proceedings. IEEE Press, May 2009.

[14] G. Theraulaz, E. Bonabeau, and J. Deneubourg. The Origin of Nest Complexity in Social Insects. *Complexity*, 3:15–25, 1998.

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

[16] Peter H. Welch and Frederick R.M. Barnes. A CSP Model for Mobile Channels. In P.H. Welch, S.Stepney, F.A.C. Polack, F.R.M. Barnes, A.A. McEwan, G.S. Stiles, J.F. Broenink, and A.T. Sampson, editors, *Communicating Process Architectures 2007*, volume 66 of *Concurrent Systems Engineering Series*, pages 17–33, Amsterdam, The Netherlands, September 2008. IOS Press. ISBN: 978-1-58603-907-3.

[17] 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.

[18] P.H. Welch and F.R.M. Barnes. Mobile Barriers for occam-pi: Semantics, Implementation and Application. In J.F. Broenink, H.W. Roebbers, J.P.E. Sunter, P.H. Welch, and D.C. Wood, editors, *Communicating Process Architectures 2005*, volume 63 of *Concurrent Systems Engineering Series*, pages 289–316, Amsterdam, The Netherlands, September 2005. IOS Press. ISBN: 1-58603-561-4.