Statistical Performance Analysis of an Ant-Colony Optimisation Application in S-NET

Kenneth MacKenzie
Philip K.F. Hölzenspies
Kevin Hammond
School of Computer Science
University of St Andrews, UK
kwxm@inf.ed.ac.uk,
{pkfh, kh}@st-andrews.ac.uk

Raimund Kirner
Nguyen Vu Thien Nga
Rene te Boekhorst
University of Hertfordshire
School of Computer Science
Hatfield, UK
{r.kirner,v.t.nguyen,
r.teboekhorst}@herts.ac.uk

Clemens Grelck
Raphael Poss
Merijn Verstraaten
University of Amsterdam
Institute of Informatics
Amsterdam, Netherlands
{c.grelck,r.poss,
m.e.verstraaten}@uva.nl

Abstract

We consider an ant-colony optimsation problem implemented on a multicore system as a collection of asynchronous stream-processing components under the control of the S-NET coordination language. Statistical analysis and visualisation techniques are used to study the behaviour of the application, and this enables us to discover and correct problems in both the application program and the run-time system underlying S-NET.

1. S-NET: Language and run-time system

S-NET is an asynchronous stream coordination language [5, 6]. It combines so-called *boxes*, which are stateless computational kernels written in any programming language, into networks that transform a single input stream into a single output stream. A stream is a potentially infinite sequence of non-overlapping, discrete data items, called *records* (or *messages*). Records are collections of named fields containing values in the box language, together with tags which contain integer values. Values from the box language are opaque to S-NET, but integer tags are visible to both the box language and S-NET. The type of a record is the set of all the names and tag-values therein.

Disjoint paths can be constructed by using combinators such as *parallel composition*, which splits the input stream into a number of streams that are fed to the operands of the parallel composition. The transformed output streams are merged to make the resulting network Single In, Single Out (SISO) again. Which record is fed to which operand is determined by the record's type. Boxes have specified input and output types, i.e. sets of names and tag-values that are expected to be in the relevant records. Where a stream is split, records are routed to that path which has the *strongest matching* input type, i.e. all names and tag-values in the input type are in the record and there is no path with an input type with more names and tag-values that are all in the record.

FD-COMA 2013 2nd HiPEAC Workshop on Feedback-Directed Compiler Optimization for Multi-Core Architectures, part of the 8th International Conference on High-Performance and Embedded Architectures and Compilers, Berlin, Germany, January 21-23, 2013
ACM SIGPLAN conference style.

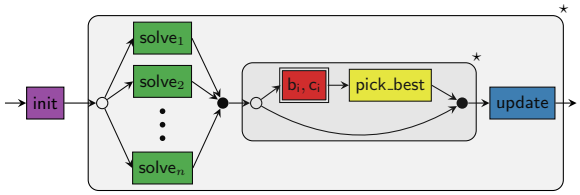
Statefulness is introduced by *synchrocells*. A synchrocell is also a SISO stream transformer, defined by a list of record types. For each record type, there is a corresponding (initially free) 'slot' in the synchrocell. In every slot, one record of that type can be stored, at which point the slot is filled. When all slots are filled, the synchrocell *syncs*, i.e. the records stored in all the slots are combined into a single record. This combined record is produced on the output of the synchrocell. Every slot of a synchrocell can only be filled once, so after a sync, the synchrocell 'dies'.

Finally, S-NET has combinators for feedback (where the output of a network is fed back to its input stream, if it does not match a specified type) and recursion (where the output of a network is fed to a new instance of that same network, if it does not match a specified type). The latter is used in the application discussed in this paper. It is referred to as the star-combinator. The operands of the star-combinator are often informally referred to as 'starred networks'. For an extensive treatise we refer to [7].

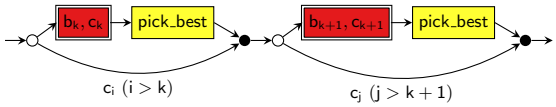
S-NET programs are compiled into binary and executed by the S-NET runtime system [4], which in turn uses the *Light-weight Parallel Execution Layer* (LPEL, [10]) for scheduling, placement and low-level thread management. User boxes as well as components that implement S-NET's coordination are all instantiated as *tasks* in LPEL. This instantiation is *ad-hoc*, in the sense that tasks are created for boxes when the network in which they occur is first reached by a record. When the run-time system determines that a box can no longer be reached by any more records, the corresponding tasks are automatically garbage collected [3].

An LPEL task is input-buffered, i.e. every task has a (bounded) input FIFO buffer into which other tasks can write and from which only the owner-task can read. A task is *enabled* when there are items in its input buffer, and it is blocked when trying to write output to a full receiving buffer. LPEL creates a *worker* for every available processor core (or a user-specified number of cores to be used). A worker is assigned tasks, and the worker's enabled tasks are executed in a round-robin fashion. A running task is never preempted; only when a task finishes or when it blocks on trying either to read from an empty input buffer or to write to a full receiving buffer, can a worker execute a different task.

Each worker has its own scheduler, and these cooperate to select tasks which are ready for execution and to execute them on the relevant worker. In the current implementation, all system tasks (synchrocells and various system boxes performing administrative tasks such as merging records) are allocated over cores in a round-robin fashion. There are also separate round-robin schedules for the



(a) Application structure



(b) Unrolled collection

Figure 1: Ant-colony optimisation application

instantiation of each user box: in the example described later, there would be one round-robin schedule for the boxes named solve, another for the pick_best boxes, and so on. See [10, 5.4.2] for further details.

2. Program structure

The application analysed in this paper is an S-Net implementation of an ant-colony optimisation algorithm [2] for the Single Machine Total Weighted Tardiness Problem (SMTWTP), which is known to be NP-hard [8]. For a detailed discussion of the functional behaviour of the application, see [1]. See also [9] for more information about the specific techniques used in the application. For the purposes of this paper, a brief introduction of the structure of the application (see Figure 1a) suffices.

The application is initialised (in the init box) with a naive, straightforward solution (stored in a record under field name b_0). This solution is marked as 'best' (thus far) and fed as input to each of n concurrently-executing 'ants' (solve tasks). Each of these generates a perturbation of the solution it is given using (bounded) randomness to create variation from other ants, and then outputs the new candidate solution r_i. A starred network then consumes the ants' outputs, and selects the best candidate to act as input for a new generation of ants. A synchrocell (denoted by ⟨sync⟩) waits for the first candidate c_1 to arrive and amalgamates it with b_0 to form a new record which is fed to a pick_best task (note that the concurrently-executing tasks need not finish in order, so that c_1 may not be equal to s_1; however, the c_i form some permutation of the s_i). The pick_best task then decides which of b_0 and c_1 is better, and outputs the result as a new record b_1, which is then combined with the next available candidate c_2 in a new synchrocell, and so on. The recursion of the starred network terminates when a record containing the name b_n is produced with the best solution for all ants of the current generation. This part of the network follows a common S-Net design pattern to mimic a finite state machine; more information on this pattern can be found in [3].

Finally, update generates inputs for the n ants of the next generation, by copying the best solution of this generation. It also updates a data structure (the *pheromone matrix*) stored in shared memory which contains heuristic information which is read by the solve tasks and used to direct their search for new solutions.

The outer box drawn in Figure 1a is a starred network that unfolds for a fixed number of generations (supplied to the application as an input parameter max_it) to run the whole cycle repeatedly and eventually produce a hopefully close-to-optimal solution to the problem.

3. Statistical Analysis and Visualisation

3.1 Structure of the data

We performed a number of runs of the ant-colony optimisation application on a 48-core server with 4 sockets each having 2 by 6 core AMD Opteron 6174 processors. We have log data for all combinations of the following parameters:

- Number of ants: 1, 10, 20, 30, 40, 50, 60
- Number of cores used for execution: 4, 8, 16, 32, 48
- Size of input problem: 200, 400, 600, 800
- Number of iterations (max_it): 100, 500, 1000, 2000

This gives a total of $7 \times 5 \times\ 4 \times 4 = 560$ datasets. In this section we will concentrate on the data with 48 cores, 100 iterations and input size 400, with the number of ants varying. We report on the data for a single run, but repeated runs with the same configuration give very similar results.

Each dataset consists of a directory containing a map file describing the assignment of tasks to workers (i.e., cores), together with one log file for each worker, listing the events that have occurred on that worker. For the 1-ant example, the data totals 448 kilobytes: the map file contains 1408 lines and the log files contain approximately 180 lines each. For the 60-ant example, the data totals 23 megabytes: the map file contains 36808 lines and the log files contain approximately 10500 lines each.

For analysis and visualisation, we use the R system [11]; the plots in this paper were produced using the R library ggplot2 [12]. The log files contain a great deal of data and can be very large, so we use a parser written in Haskell to extract information of interest and output it in a tabular form suitable for input to R.

3.2 Analysis

We are primarily interested in the statistical properties of the application with a view to predicting execution time based on input. However, the data turned out to have some puzzling features which led to the discovery of problematic issues with respect to both the S-Net/lpel system and the implementation of the ant-colony application.

Given the structure of the application, one would expect the solve boxes (i.e., the ants) to have the largest latency. Internally, each ant contains a loop with 100 iterations, whereas the other user boxes contain mostly straight-line code. Moreover, there should be little variation in the latencies of ants. Each ant is executing identical code; there is a stochastic component (each ant occasionally attempts to improve its current solution by performing a slight random perturbation) but this should average out over a single execution.

However, this is very definitely not what happens. Figure 2 shows a graph of box latencies plotted against start time for 40 ants ("latency" here refers to the time from first input to final output). Points are coloured according to the type of box whose latency is being plotted: in this case, green points denote ants (solve boxes). As expected, the execution time of ants is much greater than that of other boxes (with the exception of synchrocells, denoted by red points: however, these spend most of their lifetime waiting for input and perform very little computation). Unexpectedly, however, there are wide variations in the latencies of boxes. Moreover, latencies are not smoothly distributed; instead, boxes with similar start times tend to occur in small clumps with similar latencies.

What happens if we reduce the number of ants? Figure 3 shows a similar plot of tasks for the 30-ant example. Here we see that

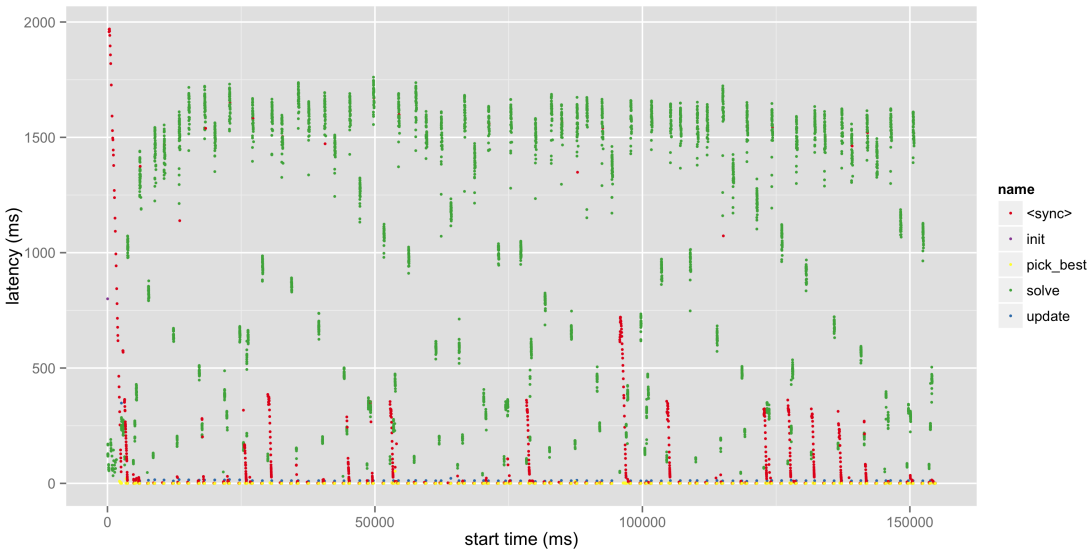


Figure 2: Task execution times for 40 ants

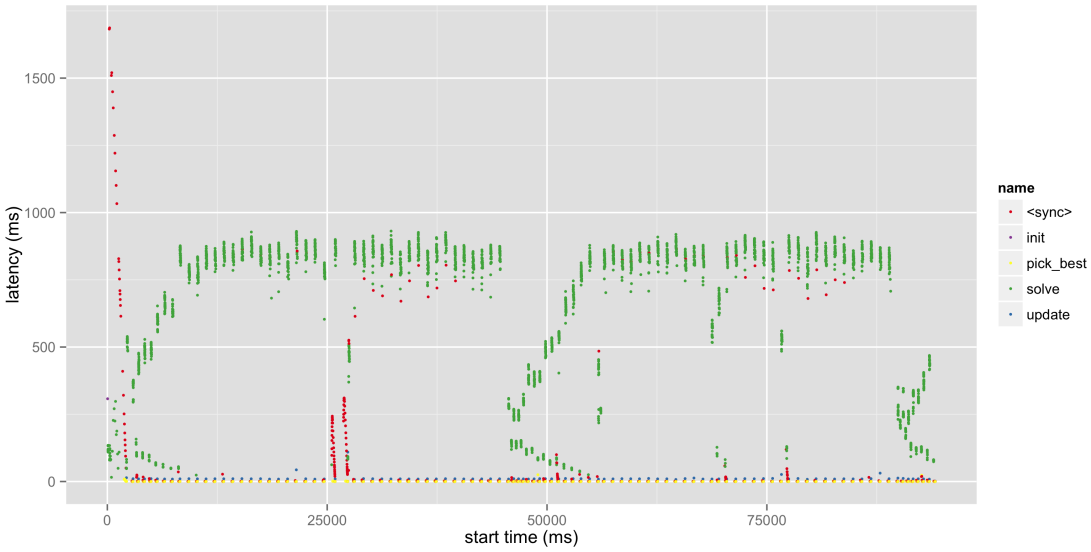


Figure 3: Task execution times for 30 ants

again latencies have a large variance; however, in this case there is a very striking periodicity evident in the variation of latencies.

For 20 ants we obtain Figure 4. In this case the latencies are much more evenly distributed and the clumping effect is less evident. Note also that the vertical scale changes, and that the latency of ants increases as the number of ants executing in parallel grows.

Table 5 shows the mean latency and the variance (taken over the entire execution of the program) for varying numbers of ants. The mean latency for 50 and 60 ants is very large, but this is to be expected since the number of ants exceeds the number of available cores (48), so some ants will have to wait for a previous ant of the same generation to finish before they can start. What is surprising is that even with 40 ants the latency is significantly higher than one would expect. The latency for the 1-ant case (15.5ms on average) can be regarded as the "true" execution time of an ant. Since the ants are supposed to be operating independently in parallel, one would expect the mean latency in the 40-ant case to be similar (perhaps with a little overhead), but instead it is more than 68 times greater.

How can this be explained? We were able to make some progress by looking more closely at execution times for ants in a single generation. In Figures 6 and 7 we display a close-up view of the plot of latencies against start times for generations 22 and 23 of the 40-ant data (the horizontal and vertical scales are the same for both plots). We see that these fall into two distinct classes. In the first class, which we will call *Type I generations*, all the ant executions start at approximately the same time and have a high latency. In the second class (*Type II generations*), the ant tasks for a

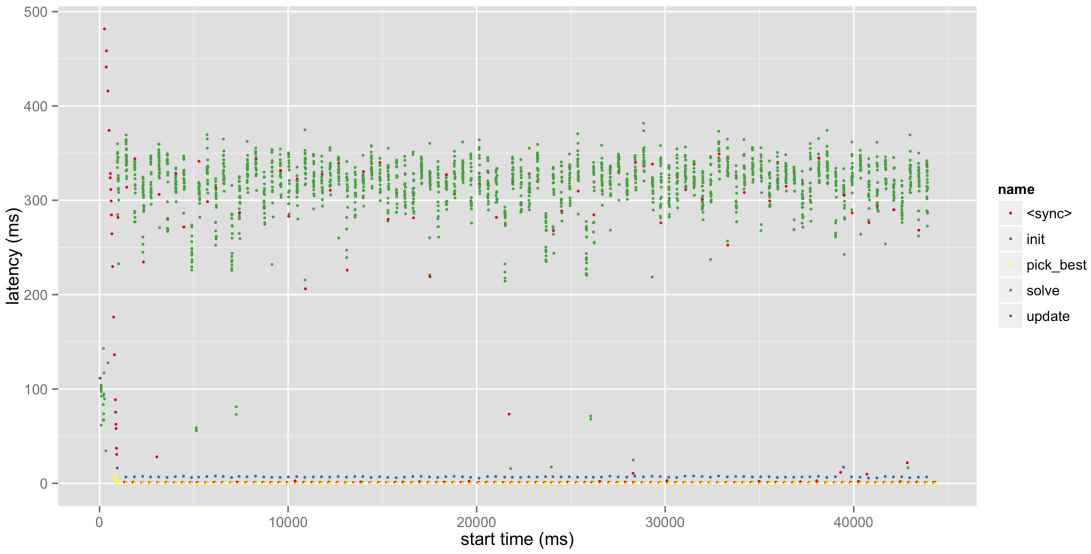


Figure 4: Task execution times for 20 ants

Ants per generation	Total number of ants	Mean latency (ms)	Standard deviation	Total execution time (s)
1	100	15.5	7.95	2.56
10	1000	126.92	18.20	18.73
20	2000	314.06	38.93	44.30
30	3000	684.21	250.75	94.15
40	4000	1059.77	552.81	154.70
50	5000	1458.28	862.37	225.82
60	6000	1441.133	849.78	315.85

Figure 5: Latency statistics for varying amounts of parallelism

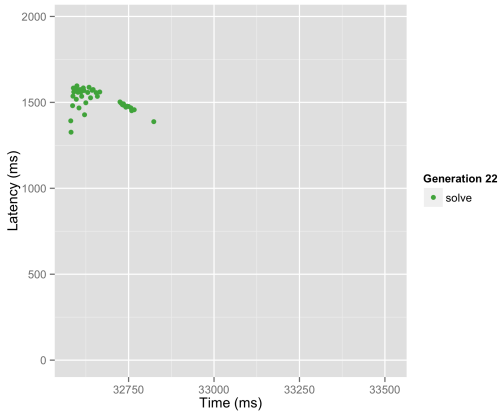


Figure 6: Ant execution times for generation 22

generation are split into two batches: each ant in the first batch has a high latency, and each ant in the second batch begins much later, but has a lower latency. Furthermore, the ants in Type II generations all have lower latencies than the ones in Type I generations.

Examination of all 100 generations shows that similar patterns are repeated throughout, with Type II generations forming about one third of the total. Moreover, there is an approximate periodicity:

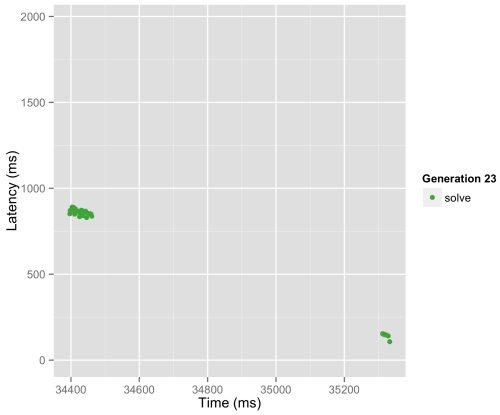


Figure 7: Ant execution times for generation 23

the generations typically (but not always) form groups of three, with one Type II generation followed by two Type I generations.

A similar dichotomy appears in the 30-ant case. The first few generations are of Type II, but then there is a long period where the majority of generations are of Type I; about halfway through, the generations revert to Type II and then the pattern repeats (cf Figure 3). In the 20-ant case, the majority of generations are Type I, with only about 7 out of 100 being Type II. For 10 ants the dichotomy disappears: all generations are of Type I.

3.3 Message analysis

The phenomena described in the previous section are still difficult to explain, but a third type of plot is very helpful. Recall that data is transferred between boxes by means of records (also called *messages*). The plots in Figures 8 and 9 show the messages emitted by boxes and the executions which they trigger.

In these figures, each horizontal line represents one message and consists of two contiguous sections (some of the lines are very short, and may be difficult to make out). The left-hand section

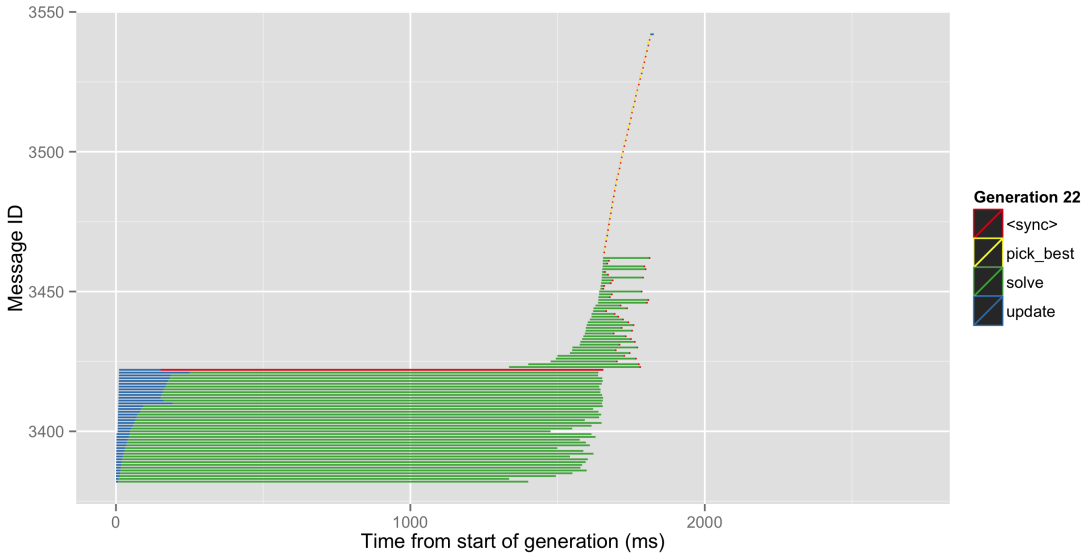


Figure 8: Messages for generation 22

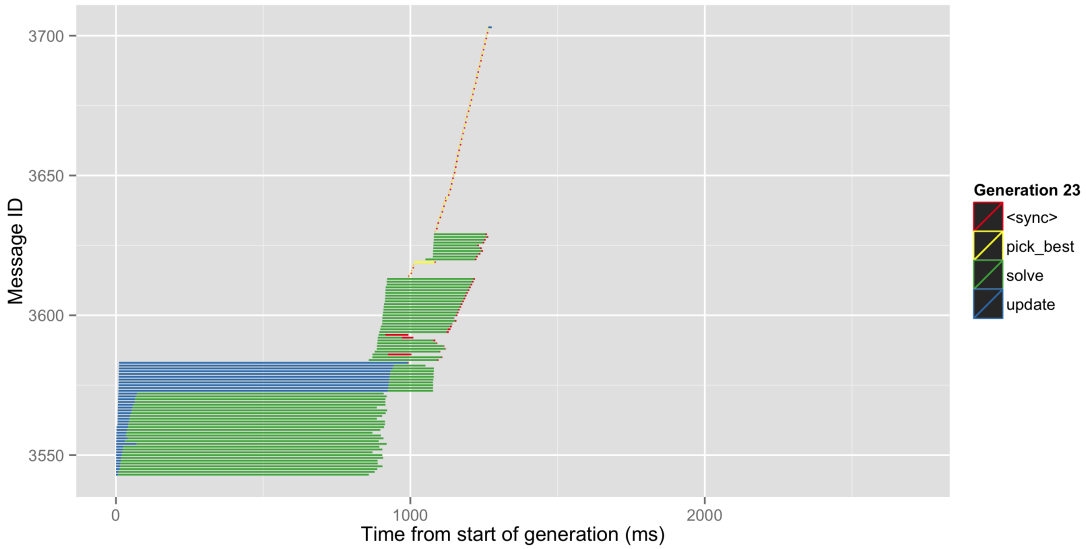


Figure 9: Messages for generation 23

represents the transmission time of a *message* M, from output to input: the colour corresponds to the type of task from which M has been output. The right-hand section represents the lifetime of the *task* T whose execution is triggered by M, from the time when T inputs M the time when T produces its final output: the colour denotes what type of task T is.

Figure 8 shows the messages for generation 22 (which is Type I), and corresponds to the plot in Figure 6. The lower section of the graph shows messages being output by the update task and received by the solve tasks. The string-like upper section shows the sequence of ⟨sync⟩–pick_best executions forming the fold operation which chooses the best result from the current execution, and the middle section shows the output messages of the solve tasks travelling to synchrocells. There is some waiting time here because the synchrocell may not be created until preceding entries have been processed in the fold.

Returning to the bottom section of Figure 8, we see that the start times of the ant tasks are slightly staggered: this is due to the ants waiting for their input records, which are output sequentially by the update task. Apart from this though, the ants all run concurrently (confirming Figure 6) and finish at roughly the same time.

On the other hand, Figure 9 (corresponding to Figure 7) depicts generation 23, of Type II. We again see the two distinct batches of ants which appeared in Figure 7, but now it is clear that *none of the ants in the second batch start to execute until after most of the ants in the first batch have finished*.

This suggests that in Type II generations some of the solve tasks are being blocked, and indeed this turns out to be the case. Recall

that the update task outputs 41 records, and that each of these forms the input to a different subsequent task: 40 of them go to new solve tasks (in numerical order), and the final one is supplied to a ⟨sync⟩ as the seed for the fold. This routing is performed by a system task called a *splitter*, which runs on a worker of its own. Close examination of the log files confirms that generations become split into two different batches when the splitter is scheduled on the same worker as one of the solve tasks. The splitter sends a number of inputs to new solve tasks and can then become blocked because some solve tasks have not yet been scheduled (or because the update task has not yet produced all of its outputs), and at this point the solve task on the splitter's worker can start to execute if its input has been emitted by the splitter. If this happens then the splitter is unable to emit its remaining outputs until the solve task has completed, and thus the remaining solve tasks are in turn blocked. This behaviour is a consequence of the scheduling strategy described at the end of §1, which purposely allows a system task to be scheduled on the same worker as a user task.

3.4 Periodic phenomena

A closer examination of the placement also explains the very distinct cyclic behaviour seen in the 30-ant example (Figure 3) and the less evident period-3 cycles in the 40-ant example.

In the 30-ant example, in generation n (counting from 1), the ants are scheduled on successive cores (counting from 0) starting at $30n + 20 \pmod{48}$. The splitter is situated on core $31n + 20 \pmod{48}$, so it follows that the splitter is n cores after the first ant (modulo 48). Thus at the start of the execution the splitter is on the same core as a low-numbered ant, leading to a high probability of the splitter being blocked, with a consequent delay to later ants. Thus initially most generations will be Type II, with all the ants having relatively low latency. As the execution progresses, the splitter moves on to clash with higher-numbered ants (or to be situated on cores without ants), leading to (a) a lower probability of blockage, and (b) fewer ants becoming blocked. Thus later generations will mostly be Type I, with all ants having high latency. On the 48th generation the splitter returns to the same core as the first ant, and the pattern repeats. This explains the cyclic nature of Figure 3.

In the 40-ant case, the first ant of generation n is situated on core $40n + 10$ and the splitter is situated on core $23n + 28$ (both modulo 48). Thus the relative position of the splitter is $23n + 28 - (40n + 10) = -17n + 18 \equiv 31n + 18 \pmod{48}$. The first few elements of this sequence are **1**, 32, **15**, 46, 29, **12**, 43, 26, **9**, 40, 23, **6**, 37, 20, **3**, 34, **17**, **0**, ... We have emboldened "low" positions (arbitrarily chosen as those less than 20) which are likely to lead to Type II generations, and we see that these form a subsequence which is roughly periodic with period 3; this reflects the structure which we saw earlier for the Type I and II generations in the 40-ant case.

We see that the very neat situation in the 30-ant case is essentially due to a numerical coincidence involving the operation of the scheduling algorithm, and that things are much more irregular for 40 ants. These features are very dependent on the details of the application. For example, in the 30-ant case the splitter is situated on core $31n + 20$; the number 31 occurs because there happen to be $126 = 2 \times 48 + 30$ other system tasks scheduled between successive ⟨split⟩ tasks. A small change in the S-Net network could remove some of these system tasks or insert new ones, leading to a significant change in the way the splitter moves about the cores and thus to a corresponding change in the latency of the ants. In particular, if the splitter was always situated on the same core as the first ant then the splitter would become blocked when the first ant started, and we would always see (short, fast) Type II generations; on the other hand if the splitter was just one core before

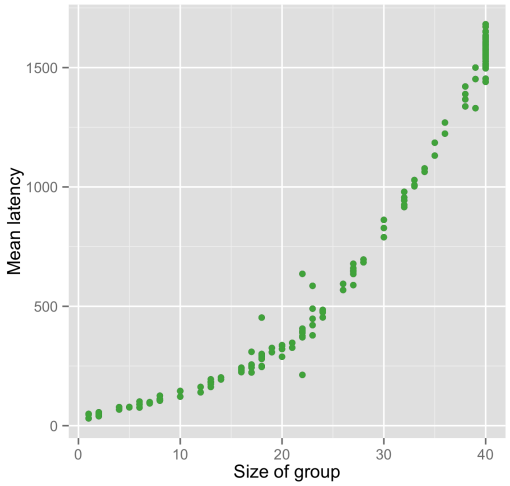


Figure 10: Mean latency of ant groups

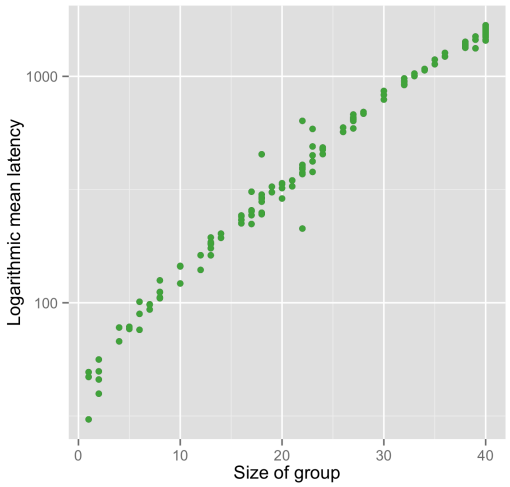


Figure 11: Logartihmic mean latency of ant groups

the ants then it would never become blocked and we would always have (long, slow) Type I generations. In more complex applications the structure of the S-Net network can evolve under program control since the number of serial and parallel replications can depend on tag values in the output of user boxes. This could lead to sudden and completely unpredictable scheduling clashes, with corresponding drastic changes in network throughput. This strongly suggests that we should strive to eliminate such behaviour; possible solutions will be discussed later in the paper.

3.5 Variation of ant latency

We have now managed to explain why the executions of the ants in a single generation can become split into two batches, but we have not yet explained the observation that average latency increases with the number of concurrent ants. This is evident in Figure 8, where the strip representing ant executions is considerably longer than in Figure 9. We can get a better idea of the dependence by partitioning the ant executions into groups which are executing concurrently and then plotting the average execution time of the

ants in the group against the size of the group. This is shown in Figure 10 for the 40-ant data.

We see that mean execution time increases rapidly with the number of concurrent ants, and the logarithmic plot in Figure 11 is close to a straight line, showing that the growth is approximately exponential.

This suggests that there is some contention between concurrent ants, and examination of the source code for the boxes confirms that this is the case. One source of contention is that at the start of execution, each solve box allocates some memory for the storage of temporary data structures: this is done using the libc library function malloc, which (in some versions of libc at least) performs a locking operation to preserve the integrity of the heap during memory allocation. However, there is another problem which is more serious. We mentioned earlier that the ant-colony optimisation method contains a non-deterministic step. In the implementation which we have been studying this is achieved by calls to libc's rand function:

```
for (k = 0; k < num_jobs-1; k++){
    ...
    q = ((double)rand())/RAND_MAX;

    if (q < const_q0) {
       ...
    }
    else{
       q = ((double)rand())/RAND_MAX;
       ...
   }
  ...
}
```

Here const_q0 is defined to be 0.9, and num_jobs is the number of jobs in the input data. In the present case, num_jobs is equal to 400, and it follows that the execution of each solve box calls rand about 440 times. However, rand is not a pure function: it contains some internal state which is preserved by a global mutex. When we have 40 ants running concurrently with each ant making over 400 calls to rand, this leads to a significant amount of contention, nullifying much of the supposed advantage of parallelism. This also explains why average ant latency increases as the number of ants increases: the more ants we have, the more contention, and hence the longer the ants take to finish. The use of rand is quite a serious error in the code (and indeed is as a violation of the S-NET "contract" which should be satisfied by box code in order to obtain valid S-NET applications), but is very easily overlooked.

3.6 Summary

Thus we have a putative explanation for the complex statistical behaviour of the ant-colony application. We have *two* interacting bugs, one relating to the behaviour of splitters in the LPEL implementation, and the other to the use of calls to library functions which perform locking operations. The first bug leads to ant generations sometimes becoming split into two subgroups which execute consecutively, and the second leads to an increase of ant latencies as the size of a group of concurrently-executing ants increases. As we have seen, this increase is exponential (Figure 10), which leads to a significant decrease in performance.

3.7 Amelioration

How can we overcome these problems? In one way or another we must re-engineer some aspects of the S-NET/LPEL runtime system. One approach could be to fuse splitters with the user boxes whose output they are distributing. Another tactic would be to give splitters a high priority, allowing them to preempt box tasks. This would also require introducing preemption into the so far collaborative task management layer of LPEL. A third approach could be

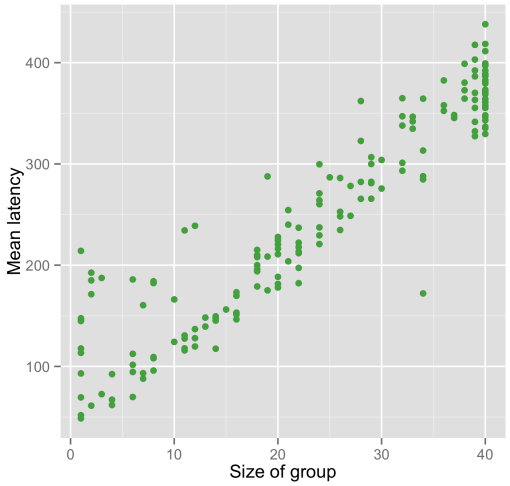


Figure 12: Latency of ant groups for revised program

to alter the scheduling algorithm so as to place splitters (and perhaps other system tasks) on a subset of cores which is disjoint from the ones running user boxes. We are in the process of experimenting with these approaches, but at the time of writing performance data for the ant-colony example was not quite available; however, we are hopeful that this will solve our problem.

The bugs in the actual implementation of the ant-colony example are more easily dealt with. We have modified the ant code by replacing malloc-allocated heap memory with stack-allocated arrays, and by replacing the calls to the libc rand function with calls to a random number generator which maintains its state locally. We have some data for this version of the application, and the performance has improved markedly: for example, the total execution time for the 40-ant case drops from 154.7 seconds to 68.9 seconds. However, it appears that there is still some contention.

Figure 12 plots average latency per group versus group size for the modified program with 40 ants, and corresponds to Figure 10 for the unmodified program (but note that the vertical scale is different). We see that the exponential behaviour shown by the original program has been replaced by linear behaviour, and that the average latency has decreased considerably. However, ideally we would expect that ant execution time should be independent of the number of ants executing concurrently, and should be close to the 15.5ms average seen in the case of a single ant. Here the latency of an ant can be of the order of 400ms, still 25 times larger than we would expect. We have as yet been unable to explain this. We believe that all calls to stateful library functions have now been eliminated from the solve boxes, and thus we should have removed any possibility of contention; however the graph indicates that contention remains. A possibility is that there is some contention in the logging system as individual cores write data to files (and in particular to the map file which records information for all cores). We intend to try some experiments to see if this might possibly be the case, for example by turning off logging for some subset of the cores and seeing if behaviour of the overall system improves.

4. Conclusion

We have used statistical and visualisation techniques to investigate the behaviour of a complex multicore application. The LPEL logging system produces a great deal of output, and it is very difficult to interpret the raw data; visualisation has helped us to gain a much better understanding of the behaviour of our application and to dis-

cover (and make progress towards correcting) not only bugs in the program we have been analysing, but also previously-unnoticed issues in the S-NET/LPEL platform itself. Thus we believe that techniques such as the ones discussed here can be helpful both to application programmers and to system developers.

Our initial motivation for these investigations was to gather statistical data relating to latency and throughput of S-NET applications with a view to providing guarantees to end-users that applications will perform within specified parameters. This has proven to be more difficult than anticipated due to the complex nature of the performance data, but we hope that our investigations will now lead to simplifications in the system which will in turn lead to better programs with more consistent behaviour, enabling us to meet our original goals.

In the short term, we wish to fully resolve the issues discussed earlier in this paper; in the longer term, we will investigate a number of other S-NET applications, and we hope that the methods developed and the experience gained in the research discussed above will enable us to make rapid progress in our analysis, and also help us to recognise and correct errors and inefficiencies in the applications which we study.

Acknowledgements

The work has been funded by the EU FP-7 project ADVANCE (Asynchronous and Dynamic Virtualisation through performance ANalysis to support Concurrency Engineering, project no. 248828).

References

[1] W. Cheng, F. Penczek, C. Grelck, R. Kirner, B. Scheuermann, and A.Shafarenko. Modeling streams-based variants of ant colony optimisation for parallel systems - a dataflow-driven approach using S-Net. In *Proc. of the 7th International Conference on High-Performance and Embedded Architectures and Compilers (HiPEAC)*, 2012.

[2] M. Dorigo and T. Stützle. *Ant Colony Optimization*. Bradford Company, Scituate, MA, USA, 2004.

[3] C. Grelck. The essence of synchronisation in asynchronous data flow. In *25th IEEE International Parallel and Distributed Processing Symposium (IPDPS'11), Anchorage, USA*. IEEE Computer Society Press, 2011.

[4] C. Grelck and F. Penczek. Implementation Architecture and Multithreaded Runtime System of S-Net. In S. Scholz and O. Chitil, editors, *Implementation and Application of Functional Languages, 20th International Symposium, IFL'08, Hatfield, United Kingdom, Revised Selected Papers*, volume 5836 of *Lecture Notes in Computer Science*, pages 60–79. Springer-Verlag, 2011.

[5] C. Grelck, S. Scholz, and A. Shafarenko. Asynchronous Stream Processing with S-Net. *International Journal of Parallel Programming*, 38(1):38–67, 2010.

[6] C. Grelck, S.-B. Scholz, and A. Shafarenko. A Gentle Introduction to S-Net: Typed Stream Processing and Declarative Coordination of Asynchronous Components. *Parallel Processing Letters*, 18(2):221–237, 2008.

[7] C. Grelck, Shafarenko, A. (eds):, F. Penczek, C. Grelck, H. Cai, J. Julku, P. Hölzenspies, Scholz, S.B., and A. Shafarenko. S-Net Language Report 2.0. Technical Report 499, University of Hertfordshire, School of Computer Science, Hatfield, England, United Kingdom, 2010.

[8] J. Lenstra, A. Rinnooy Kan, and B. P. Complexity of machine scheduling problems. *Annals of Discrete Mathematics*, pages 343–362, 1977.

[9] D. Merkle and M. Middendorf. An ant algorithm with a new pheromone evaluation rule for total tardiness problems. In *Proceedings of the EvoWorkshops 2000*, number 1803 in LNCS, pages 287–296. Springer Verlag, 2000.

[10] D. Prokesch. A light-weight parallel execution layer for shared-memory stream processing. Master's thesis, Technische Universität Wien, Vienna, Austria, Feb. 2010.

[11] R Development Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2012. ISBN 3-9000051-07-0.

[12] H. Wickham. *ggplot2: elegant graphics for data analysis*. Springer New York, 2009.