Modeling Streams-based Variants of Ant Colony Optimisation for Parallel Systems

A Dataflow-driven Approach Using S-Net

Wei Cheng
SAP AG
SAP Research
Karlsruhe, Germany
wei.cheng@sap.com

Frank Penczek
University of Hertfordshire
School of Computer Science
Hatfield, UK
f.penczek@herts.ac.uk

Clemens Grelck
University of Amsterdam
Institute of Informatics
Amsterdam, Netherlands
c.grelck@uva.nl

Raimund Kirner
University of Hertfordshire
School of Computer Science
Hatfield, UK
r.kirner@herts.ac.uk

Bernd Scheuermann
SAP AG
SAP Research
Karlsruhe, Germany
bernd.scheuermann@sap.com

Alex Shafarenko
University of Hertfordshire
School of Computer Science
Hatfield, UK
a.shafarenko@herts.ac.uk

ABSTRACT

In this paper we present the implementation of a concurrent ant colony optimisation based solver for the combinatorial Single Machine Total Weighted Tardiness Problem (ACO-SMTWTP). We introduce S-Net, a coordination language based on dataflow principles, report on the performance of the implementation and compare it against a sequential and a parallel implementation of the same algorithm in C. As the workload of the optimisation algorithm is highly irregular we consider this application to be an important use-case for runtime measurement directed optimisations of the coordination rogram as much as for guiding optimisations of numerical code.

1. INTRODUCTION

ACO is a meta-heuristic inspired by the foraging behaviour of ants [5]. A number of artificial ants iteratively construct solutions to a given combinatorial optimisation problem. Ants are thereby guided by so called pheromone information that previous ants, which have found good solutions, have disposed to mark their decisions in the solution construction process. Since the artificial ants construct their solutions independently and since the core of the algorithm consists of iteratively repeated instructions, ACO is very attractive for parallel execution on multi processor architectures (for an overview see [18]). The large majority of parallel ACO implementations were presented during the last ten years. These implementations mainly follow two distinct parallelisation approaches. In the master-slave approach, a master processes global information (like updating pheromone information, or determining globally best solution) and a number number of slaves (or workers) execute subordinate tasks (like constructing solution, or calculating fitness). The other main group follows a multi-colony approach where a number of colonies search for good solutions using their own pheromone matrices while exchanging information in certain time intervals.

In this publication, we discuss dataflow-oriented variants of parallel ACO algorithms targeting multi-core systems. A new approach to modeling such algorithms as a streaming network is presented. Streaming-based ACO algorithms differ from the above-mentioned parallelisation principles as they consider a continuous (possibly non-generational) stream of artificial ants being piped through pheromone information which is spread across multiple parallel processing units in a fixed mapping (or vice versa). The presented ACO variants shall be distinguished from prior work, where the complexity of streaming-based ACO algorithms has been examined for the theoretical RMesh computing model [16], or hardware-oriented and very fine-grained variants of ACO have been implemented for FPGAs using VHDL (see e.g. [24]). Therefore the paper at hand is the only known dataflow-driven approach to modeling streaming-oriented ACO in software directly addressing and exploiting the steadily increasing degree of parallelism in present and future multicore systems. Furthermore, the coordination technology used for the implementation allows for flexibly modeling and adapting the degree of parallelism and granularity of the ACO implementation according to the target system.

ACO algorithms are of the sort that lends itself easily to parallel decomposition in two ways. First of all, the process of ant-colony optimisation naturally breaks down into constructing solutions, selecting the best one and updating the underlying pathway structure. These stages interact with each other in a pipelined manner: the output of one stage delivers the necessary data to the input of another. The second form of decomposition is into individual activities of the same kind. Those can be of greatly varying duration and can cause the computational load to drift from one locus of a distributed system to the next, creating flash points, load imbalances, congestion, etc. It is worth repeating that these issues, viz. pipeline decomposition and asynchronous stream processing are common to a range of distributed, irregular problems from molecular dynamics [21] to machine graphics [20] to various signal processing and control applications. However the case in point may be seen as somewhat more peculiar than others.

ACO is a good example of a multilevel separation of concerns. There is a general skeleton of ant-colony optimisation, but there is also a set of specific implementations that can be changed from time to time to suit different optimisation agendas. Here we have the combination of a software engineering challenge: keeping the building blocks of the application suitably encapsulated and abstracted, and a distributed computing challenge: supplying the necessary resources to those blocks and properly connecting them with sufficiently low-latency, high throughput data streams. This is where the coordination technology introduced with S-Net comes into play. S-Net [10] was designed for keeping the two concerns separate while at the same time enabling the application designer to focus on the distribution aspect of the code. Being a coordination language as well as a component technology it allows different distribution solutions to be expressed abstractly, in network form, as well as properly instrumenting the components and profiling the application run-time behaviour.

The main form of glue characteristic of S-Net is a single connecting stream. The local state of computation is embodied in a record that floats down that stream from one component to another. The receiving component has the ability to process a part of the data contained in the record while not caring and having no knowledge of any other parts, which is a property called “flow inheritance” in S-Net lingo. An output record from a component will retain a copy of such parts (hence “inheritance”), which is explained in Section 2. In contrast to OOP, this form of inheritance allows for subsequent stages of a cascaded processing scheme to access the information not affected by earlier stages *locally* thus avoiding the linkage between hierarchical and physical remoteness, characteristic of object-based solutions.

The ability of S-Net to float the local computation state, while preserving inheritance, also makes processing components virtually stateless, since any state they may require is appended to the state record without being exposed to irrelevant components (that is taken care of by the type system, which checks that only those items that are declared by a component’s type signature is ever affected by the component). Thus the state record, which can be circulated *around* a component, turns a temporal sequence of state transitions as it is observed normally within statefull objects, into a “spatial” stream of state records.

What is the advantage of such an approach? Indeed at first glance there are only disadvantages: a seemingly higher storage demand, the need to keep inherited baggage local to its consumer, while at the same time avoiding excessive copying, etc. However, these problems are well understood and satisfactory solutions are available [8]. The biggest gain of S-Net technology is its ability to place work at zero cost on a distributed system. This comes from the fact that the S-Net components are stateless and hence can be replicated at zero cost and placed anywhere, including at multiple sites in a distributed system. The real cost is, of course, storage, not primarily the storage for records, which is highly optimised by reference-counting and intelligent caching, but synchronisation storage, i.e. memory for records that need to be joined with other records yet to be produced. Again, the unique advantage of S-Net is the fact that synchronisation storage is componentised, and that those components are devoid of domain-specific semantics, being fully defined by the coordination language. It is therefore quite possible to manage, instrument and profile synchro-storage in a manner independent of the evolving code of processing components (called “boxes” in S-Net lingo).

This paper will demonstrate how separation of box and coordination concerns, inheritance-based component hierarchies and encapsulated synchro-storage help to design a more expressive, better manageable and more easily profileable ACO code. The remainder of the paper is organised as follows: Sections 2 and 3 elaborate on the S-Net coordination technology and the corresponding toolchain. Sections 4 and 5 introduce the concept of ant colony optimisation in greater detail and explain our S-Net implementation. We report on extensive experiments with this S-Net implementation of ant colony optimisation on a 48-core server system in Section 6. Section 7 sketches out some related work before we draw conclusions in Section 8.

2. S-NET IN A NUTSHELL

S-Net is a high-level, declarative coordination language based on the concept of stream processing. As such S-Net promotes functions implemented in a standard programming language into asynchronously executed stream-processing components, coined *boxes*. Both imperative and declarative programming languages qualify as box implementation languages for S-Net, but we require any box implementation to be free of state on the coordination level. More precisely, a box must not carry over any information between two consecutive activations on the streaming layer.

Each box is connected to the rest of the network by two typed streams: one for input and one for output. Messages on these typed streams are organised as non-recursive records, i.e. sets of label-value pairs. The labels are subdivided into *fields* and *tags*. The fields are associated with values from the box language domain; they are entirely opaque to S-Net. Tags are associated with integer numbers, which are accessible both on the coordination and on the box level. Tag labels are distinguished from field labels by angular brackets.

Operationally, a box is triggered by receiving a record on its input stream. As soon as that happened, the box applies its box function to the record. In the course of function execution the box may communicate records on its output stream. Once the execution of the box function has terminated, the box is ready to receive and to process the next record on the input stream.

On the S-Net level a box is characterised by a *box signature*: a mapping from an input type to a disjunction of output types. For example,

```
  box foo ((a,<b>) -> (c) | (c,d,<e>));
```

declares a box `foo` that expects records with a field labelled `a` and a tag labelled `b`. The box responds with an unspecified number of records that either have just field `c` or fields `c` and `d` as well as tag `e`. The associated box function `foo` is supposed to be of arity two: the first argument is of type `void*` to qualify for any opaque data; the second argument is of type `int` as the joint interpretation of tag values by the coordination and the box/application layer.

The box signature naturally induces a *type signature*. Whereas a concrete sequence of fields and tags is essential for the proper specification of the box interface, we drop the ordering when reasoning about boxes in the S-Net domain.

Consequently, this step turns tuples of labels into sets of labels. Hence, the type signature of box `foo` is

```
{a,<b>} -> {c} | {c,d,<e>} .
```

We call the left hand side of this type mapping the *input type* and the right hand side the *output type*. We use curly brackets instead of round brackets to emphasise the set nature of types.

To be precise, this type signature makes `foo` accept *any* input record that has *at least* field `a` and tag `<b>`, but may well contain further fields and tags. The formal foundation of this behaviour is *structural subtyping* on records: Any record type t_1 is a subtype of t_2 iff $t_2 \subseteq t_1$. This subtyping relationship extends to multivariant types, e.g. the output type of box `foo`: A multivariant type x is a subtype of y if every variant $v \in x$ is a subtype of some variant $w \in y$.

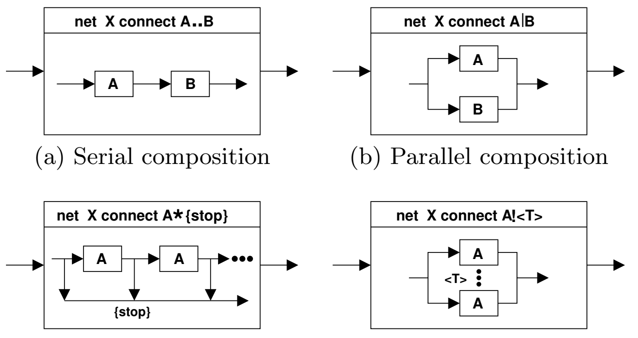
Subtyping on input types of boxes raises the question what happens to the excess fields and tags. S-Net supports the concept of *flow inheritance* whereby excess fields and tags from incoming records are not just ignored in the input record of a network entity, but are also attached to any outgoing record produced by it in response to that record. Subtyping and flow inheritance prove to be indispensable features when it comes to make boxes that were designed in isolation collaborate in a streaming network.

It is a distinguishing feature of S-Net that it neither introduces streams as explicit objects nor that it defines network connectivity through explicit wiring. Instead, it uses algebraic formulae to describe streaming networks. The restriction of boxes to a single input and a single output stream (SISO) is essential for this. S-Net provides four network combinators: static serial and parallel composition of two networks and dynamic serial and parallel replication of a single network. These combinators preserve the SISO property: any network, regardless of its complexity, is an SISO entity in its own right.

Let `A` and `B` denote two S-Net networks or boxes. Serial combination (`A..B`) constructs a new network where the output stream of `A` becomes the input stream of `B`, and the input stream of `A` and the output stream of `B` become the input and output streams of the combined network, respectively. As a consequence, `A` and `B` operate in a pipeline.

Parallel combination (`A|B`) constructs a network where incoming records are either sent to `A` or to `B` and the resulting record streams are merged to form the overall output stream of the combined network. The type system controls the flow of records. Each operand network is associated with a type signature inferred by the compiler. Any incoming record is directed towards the operand network whose input type is better matched by the type of the record. If both operand network's input types are matched equally well, one alternative is selected non-deterministically. Parallel composition can be used to route different kinds of records through different branches of the network (like branches in imperative languages) or, in the presence of subtyping, to create generic and specific alternatives triggered by the presence or the absence of certain fields or tags.

The parallel and serial composition combinators have their infinite counterparts: serial and parallel replication combinators for a single operand network. The serial replication



(a) Serial composition (b) Parallel composition (c) Serial replication (d) Parallel replication

Figure 1: Illustration of the four S-Net network combinators

combinator `A*`*`type`* constructs an infinite chain of replicas of `A` connected by serial combinators. The chain is tapped before every replica to extract records that match the type specified as the second operand. More precisely, the type acts as a so-called *type pattern*; pattern matching is defined via the same subtype relationship as defined above. Hence, a record leaves a serial replication context as soon as its type is a subtype of the type specified in the type pattern. The parallel replication combinator `A!<`*`tag`*`>` also replicates network `A` infinitely, but this time the replicas are connected in parallel. All incoming records must carry the tag `<tag>`. This tag's value determines the network replica to which a record is sent.

In practice, we often see boxes that mostly or entirely serve housekeeping purposes, such as renaming, duplication or elimination of fields and tags or simple arithmetic operations on tag values. While all this can be easily accomplished using a user-implemented box, it is often more convenient to do this housekeeping on the S-Net level as it directly affects network construction. The construct we introduce for these purposes is called a *filter* and it looks as follows:

$$[pattern \rightarrow record_1; record_2; \ldots record_n] .$$

The type pattern on the left is a set of labels while each of the record specifiers on the right defines the output. For example, the filter

```
[{a, b, <c>} -> {a, z=a, <t>};
                {b, a=b, <c=c+1>}]
```

consumes a record with fields `a`,`b` and the tag `c` and creates two new records: The first record has field `a` with the original value, field `z` with the same value and a tag `<t>` set to zero. The second record has fields `b` with the original value, `a` with the same value as `b` and the tag `<c>`, whose value is incremented by 1:

While any box or filter can split a record into parts, we so far lack means to express the complementary operation: merging two records into one. For this purpose, S-Net features dedicated *synchrocells*. A synchrocell has the syntactic form `[|`*`type`*`,`*`type`*`|]`. Similar to serial replication the types act as patterns for incoming records. A record that matches one of the patterns is kept in the synchrocell. As soon as a record arrives that matches the other pattern, the two records are merged into one, which is forwarded to the output stream. Incoming records that only match previously matched patterns are immediately forwarded to the output stream. Hence, a synchrocell becomes an identity after successful synchronisation and may be removed

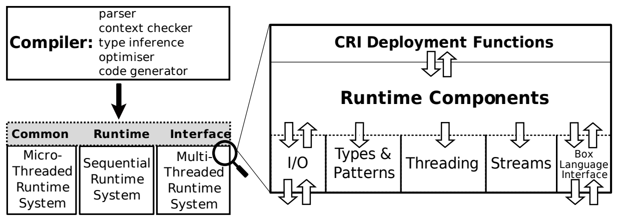


Figure 2: S-Net toolchain architecture

by a runtime system. The extremely simplified behaviour of synchrocells captures the essential notion of synchronisation in the context of streaming networks. More complex synchronisation behaviours, e.g. continuous synchronisation of matching pairs in the input stream, can easily be achieved using synchrocells and network combinators. Details can be found in [7].

To summarise S-Net is an abstract notation for streaming networks of asynchronous components. It is a notation that allows programmers to express concurrency in an abstract and intuitive way without the need to reason about the typical annoyances of machine-level concurrent programming. Readers are referred to [8, 10] for a more thorough presentation of S-Net and to [11, 19] for other case studies on application programming with S-Net.

3. S-NET TOOL CHAIN

Fig. 2 illustrates the multi-layered architecture of the S-Net toolchain. The S-Net compiler `snetc` reads S-Net source code and checks it for lexicographic and syntactic correctness. The core of the compiler is the type inference system that associates any combinator subexpression with an S-Net type that defines routing of records through the network. The S-Net compiler emits fairly high-level C code with calls to S-Net library functions, the *common runtime interface*. In fact, the C code still very much resembles the original S-Net source code with individual library functions implementing boxes, filters, synchrocells and the network combinators, properly parameterised for the concrete application.

The common runtime interface features several implementations; in this paper we focus on the *multithreaded runtime system* [9]. It consists of two layers that are mutually dependent: the *deployment layer* sets up a system of asynchronous components communicating via bounded buffers in shared memory. At this level, combinators are resolved into split and merge style components. The *component layer* implements the dynamic behaviour of the various S-Net components. Both layers depend on each other as networks with replication combinators dynamically evolve.

The component layer of the runtime system is based on a number of separate auxiliary modules that manage records, implement type pattern matching, realise streams as buffers and, last not least, control the interfacing of S-Net with the outside world on the global begin and end of the streaming network and towards the box languages used to implement the boxes. The most important of these modules is the *threading layer* that controls low-level thread management. We currently have two implementations of the threading layer: one maps each S-Net component to its own Posix thread and, thus, leaves the scheduling of S-Net components to the operating system. The more elaborate threading layer implementation, named LPEL [23], actively manages S-Net components as non-preemptive tasks with low-overhead scheduling of tasks to a fixed small number of Posix threads for effective utilisation of multi-core processors. This threading layer is used in the experiments reported in Section 6; it provides ample opportunities for runtime profiling.

```
1 initialize;
2 while termination condition not met do
3     foreach ant do
4         constructSolution;
5     end
6     pickBest;
7     update;
8 end
```

Algorithm 1: Top-level structure of a typical, generic ACO algorithm

4. ACO ALGORITHM

This section briefly introduces the structure and operation of a typical sequential ACO algorithm from a generic perspective for static combinatorial optimisation problems (for a more detailed introduction to ACO refer to e.g. [5]). The description conforms with the conventional (non-streaming oriented) programming style. As shown in Algorithm 1, Line 1, the algorithm begins by initialising problem-dependent parameters (e.g. calculating evaluation parameters, setting initial pheromone values etc.). This is followed by an interative body, where a number of m ants repeatedly construct solutions (Line 4) by making a sequence of local decisions, e.g. successive selections of items in the solution vector. Every decision is made randomly according to a probability distribution over the so far unchosen items in selection set S and depending on pheromone information and heuristic information. Pheromone information is encoded in an $n \times n$ pheromone matrix $[\tau_{ij}]$. Pheromone value τ_{ij} expresses the desirability to assign an item j to place i in the solution vector. Ant decisions are further supported by problem-specific heuristic information η_{ij}.

Assuming the ant is positioned in row i of the pheromone matrix (assigning the i-th item in the solution vector), with probability q_0 the ant makes a deterministic decision and with probability $1 - q_0$ it makes a random decision: Deterministic decision: Item $j \in S$ is selected which maximises $\tau_{ij}^{\alpha} \cdot \eta_{ij}^{\beta}$. Random decision: With probability p_{ij} item $j \in S$ is selected with $p_{ij} = \frac{\tau_{ij}^{\alpha} \cdot \eta_{ij}^{\beta}}{\sum_{h \in S} \tau_{ih}^{\alpha} \cdot \eta_{ih}^{\beta}}$ and $p_{ij} = 0$ if $j \notin S$. Parameters α and β determine the relative influence of pheromone values and heuristic values.

At the end of an iteration, when m solutions have been generated, the best solution π^* of all iterations (global-best solution) is determined (Line 6) which is used to update the pheromone matrix (Line 7): $\tau_{ij} := (1-\rho) \cdot \tau_{ij} + \rho \cdot \Delta_{ij}$. Commonly, increment Δ_{ij} enforces pheromones along the trail of the best solution, i.e. $\Delta_{ij} > 0$ if $\pi^*(i) = j$ and $\Delta_{ij} = 0$ otherwise. Parameter $0 < \rho \leq 1$ models the pheromone evaporation rate. The ACO algorithm executes a number of iterations until a specified stopping criterion has been met, e.g. a predefined maximum number of iterations has been executed (Line 2).

5. STREAMS-BASED ACO

This section describes various approaches to transferring sequential ACO algorithms into parallel streaming-based variants. ACO algorithms have been chosen as example as their structure consisting of an initialisation followed by an iterative body of repeatedly executed optimisation operators is representative for a wealth of other meta-heuristics including e.g. Evolutionary Algorithms, Simulated Annealing, Tabu Search etc.

5.1 ACO in S-Net

Modelling the discussed ACO algorithm in S-Net is accomplished with minimal effort. The stages of Algorithm 1 are implemented as boxes; the control-flow of the application is straight-forwardly transformed into a data-flow representation as the resulting streaming network shown in Figure 3 illustrates. The implementation for each box is in large parts provided by existing C code as S-Net provides interfaces for external programming languages (cf. [8] for a more detailed interface description) such that for existing code (e.g. written in C) only small wrapper functions need to be supplied.

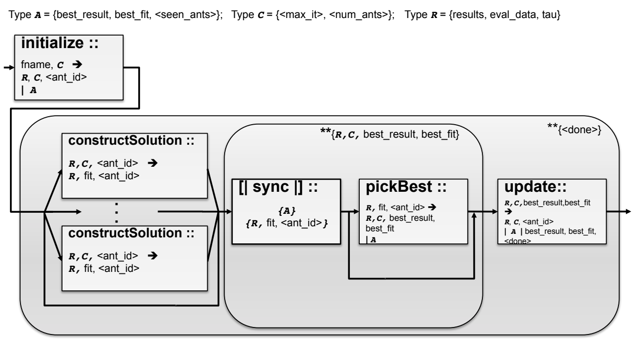


Figure 3: ACO algorithm modeled in S-Net

Assuming that the individual ant decisions are independent (which is commonly true) our S-Net implementation follows the most intuitive approach to parallelise the ACO algorithm by expressing the `constructSolution` procedures (virtual ants) as concurrent components (an outlook on further ACO streaming variants is given in Section 5.2). This is achieved by using the indexed parallel replication operator using `<ant_id>` as identification tag. The actual number of parallel ants is determined by the range of `<ant_id>` tags emitted by the `initialize` component which in turn is connected to the remaining network via as serial combinator.

The box `initialize` reads input data, initialises the optimisation problem and sends a stream of ants according to `<ant_id>` to different instances of the `constructSolution` box. The streams carry records of various parameters, such as constants C, iteratively updated best results A and iteration related variables R.

After the parallel instances of `constructSolution` finish processing the results are accumulated to identify the current best solution. This is achieved by employing a merge construction consisting of a synchro-cell and an accumulator box: The synchro-cell keeps an accumulator record of type A and joins this up with a result emitted from one of the `constructSolution` instances. The `pickBest` box adds this result to the accumulator. Through the star combinator (indicated by two stars and an exit pattern in Fig. 3) a serial chain of synchro-cells and `pickBest` instances is established. The accumulator is sent down to the next stage of the chain where it is joined up with a result of one of the remaining `constructSolution` instances by a synchro-cell. Through the `<num_ants>` tag the `pickBest` box determines when the last result has been seen; in this case the merging process ends and a record is sent on to the box `update`.

The box `update` has two choices: it can update the pheromone matrix and start a new iteration, or stop the stream and output answers if the stopping criterion is met, i.e. box `update` may end the conceptually infinite pipeline with producing a record containing the tag `<done>`.

5.2 Further Streaming Variants – An Outlook

The above-mentioned implementation in S-Net represents the probably most straight-forward idea of a streaming-oriented ACO algorithm. In principle, this implementation considers a set of m stationary ants processing a continuous stream of pheromone matrices (and problem instances) where each ant is assumed to construct and evaluate a complete solution per call to `constructSolution`. This is followed by a synchronisation before starting the comparison and update procedures. Further research in streaming-oriented ACO may respect the following variations: i) *streaming objects* – to consider the pheromone matrix as stationary unit which is traversed by a continuous stream of ants, ii) *partitioning* – to sub-divide pheromone matrices or to split ants into groups of consecutive ant decisions, iii) *evaluation* – to also compute fitness evaluations in parallel with solution construction, iv) *update* – to also execute pheromone updates in parallel with solution construction without prior synchronisation (which may lead to a non-generational update concept). For each variation, the fundamental instructions remain unchanged, to a large extend existing code can be re-used. The main challenge is to model the problem partitioning and the coordination of dataflow and synchronisation. These tasks shall be largely supported by the expressiveness of S-Net and through the de-coupling of concurrency and algorithm engineering.

6. PERFORMANCE EVALUATION

In this section, the setup and the results of the experimental performance evaluation of ACO for the Single Machine Total Weighted Tardiness Problem (SMTWTP) are presented. For a detailed introduction we refer to e.g. [4].

6.1 ACO Instantiation for SMTWTP

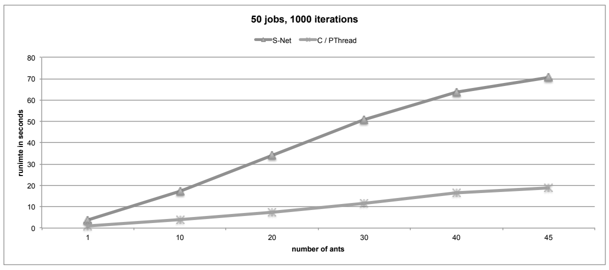
The algorithm described here instantiates the generic ACO algorithm (see Section 4) for the Single Machine Total Weighted Tardiness Problem, where n jobs (items) need to be scheduled on a single machine. Associated with each job j are its processing time p_j, weight w_j and due date d_j. The goal of SMTWTP is to find a job sequence π (solution vector represented by a permutation of job numbers $1, \ldots, n$) which minimises the total weighted tardiness $TW = \sum_{i=1}^{n} w_{\pi(i)} \cdot T_{\pi(i)}$, where $T_j = \max\{0, C_j - d_j\}$ denotes the tardiness and C_j defines the completion time of job $j = \pi(i)$. The complexity of SMTWTWP was shown to be $\mathcal{NP}$-hard [14]. For large problem instances ($n > 50$,) exact algorithms often fail to calculate the optimum in acceptable computation time [1, 3]. Therefore alternate approaches try to find good, near-optimal, solutions by applying different heuristics, amongst them ACO algorithms belong to the best performing meta-heuristics [15]. For this paper, the Apparent Urgency (AU) heuristic [22] was chosen to derive domain-specific guidance $\eta_{ij} := 1/au_j$. The AU-heuristics sorts the jobs in non-decreasing order of its apparent urgency $au_j = (w_j/p_j) \cdot \exp(-\max\{d_j - C_j, 0\}/k\bar{p})$ with $\bar{p}$ expressing the average processing time of the unscheduled jobs and k a parameter chosen as suggested in [22]. AU exhibited a competitive performance in prior evaluations [4]. For the sake of brevity, additional local search routines are disregarded in this paper. During the update process pheromone increments are calculated as $\Delta_{ij} = n/TW^*$ with TW^* denoting the total weighted tardiness of the best schedule π^*.

6.2 Experimental Setup and Evaluation

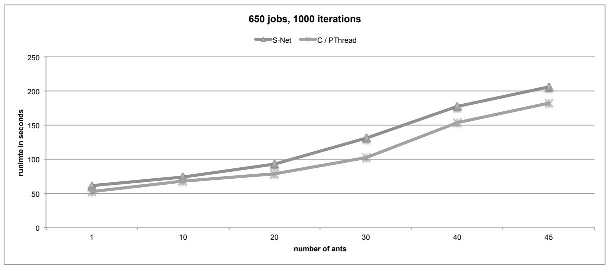
The common starting point for the comparative study was a sequential ACO algorithm for SMTWTP written in C. This code was de-composed to derive a concurrent streaming-oriented variant using S-Net as outlined in Section 5. As counterpart for the experimental evaluation, the sequential C code was manually parallelised (without streaming) using PThreads such that each ant executed in a separate thread. Both algorithms perform the same calculations such that their optimisation behaviour is identical and the performance evaluation can be restricted to a pure runtime comparison. The runtime measurements presented in this section were collected on a 48-core computation server, comprising 4 sockets with 2 by 6 core AMD Opteron 6174 processors. The machine is equipped with 256GB of main memory and runs Linux kernel version 2.6.35. The experiments have been repeated three times with stable measurements (average deviation of runtime < 1%).

A first set of experiments has been conducted to assess the overall performance of the implementation. In Fig. 4 the absolute runtimes on several problem sizes are presented. Each of the graphs shows the recorded runtime of the S-Net and C/PThreads implementation. For small problem sizes where each solving step per ant only takes a fraction of a second the results clearly show the overheads that the S-Net implementation comes with. For larger problem sizes, and accordingly longer runtimes, these overheads are less significant and the performance is closer to that of the hand-parallelised code. These overheads stem mostly from the differences in memory allocation strategies; where the C code operates on global arrays that are allocated only once in the beginning, the S-Net implementation reallocates temporary arrays in each iteration. This is owed to the state-freeness of boxes that are not allowed to share state through global variables. In addition to this, the merging phase of the S-Net implementation that iteratively collects all sub-results in an accumulator incurs higher costs in comparison to the barrier synchronisation that is used in the C implementation.

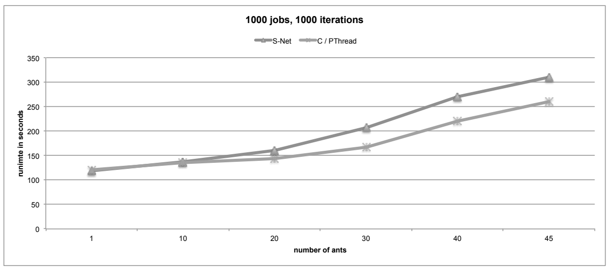
A second class of experiments was run to further quantify the efficiency of the implementations. In these experiments each implementation was given 40 inputs. From the total runtime measures, the runtime per record was computed. Due to the streaming nature of S-Nets, the S-Net implementation



(a) small problem size

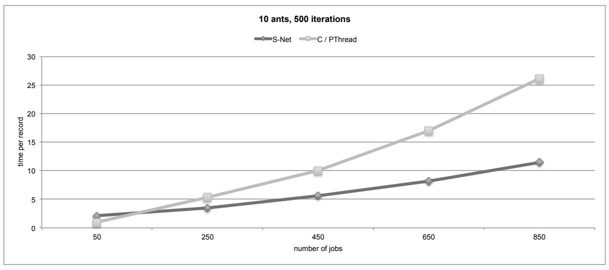


(b) medium problem size

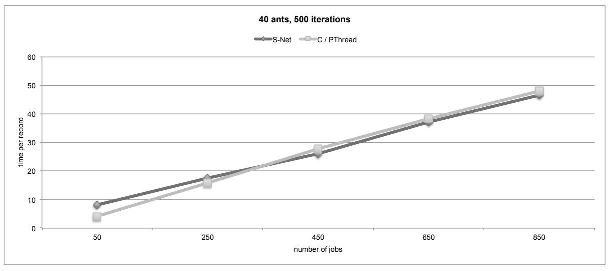


(c) large problem size

Figure 4: Absolute runtimes of the algorithm implemented in C and S-Net on various problem sizes.



(a) 10 ants on various problem sizes



(b) 40 ants on various problem sizes

Figure 5: Runtimes of the S-Net and multi-threaded C implementation for 10 ants (a) and 40 ants(b) on several problem sizes. The runtime shown is for one record computed from the runtime on input of 40 problems.

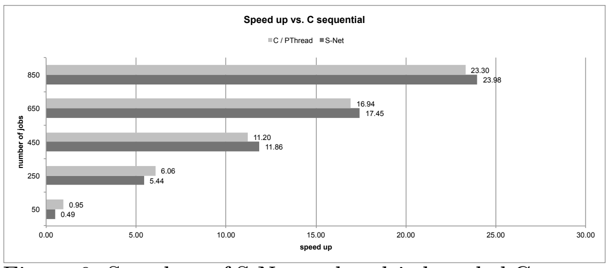


Figure 6: Speed up of S-Net and multi-threaded C compared to a sequential implementation.

is able to process the stream of input records concurrently in a pipelined fashion. As each iteration step is carried out in its own instance of the solver network, each iteration establishes a new stage of a computational pipeline in which all stages may run in parallel. In S-Net it is very simple and straight-forward to further increase the degree of exploitable concurrency by using the parallel replication combinator. If applied to the top-level network several instances of the application are deployed in parallel. The number of instances is controlled by tag values that are inserted into the input records. Each value corresponds to one instance of the replication combinator's operand. We have used values between 1 and 8 which distributed the records of the input stream round-robin to the instances of the entire network, i.e. 5 records are processed by each instance where all instances may potentially execute in parallel. Fig. 5 shows the effect that capitalising on the pipeline parallelism and the application of the split combinator to the outer-most network have. A comparison to the multi-threaded C implementation which processed the 40 problems one after the other is not fully suitable for a fair comparison as for a small number of ants the S-Net implementation is able to make use of more of the available computing resources than the C implementation. This results in considerably lower runtimes for the S-Net implementation. For larger numbers of ants for which the C implementation also claims most of the available resources the difference in runtime per record diminishes. In this setting both implementations perform very similar as shown in the second graph of Fig. 6. We include the measurements here nevertheless as they demonstrate how S-Net combinators may be used to rapidly devise experiments for various scheduling techniques to maximise exploitable concurrency.

In order to compare the multi-threaded C implementation and the S-Net implementation to the original, sequential implementation of the application Fig. 6 shows the speed-up of the parallel versions against the sequential code. These numbers are based on runtimes for 40 ants and 500 iterations on various problem sizes. As can be seen, both implementations show very similar behaviour; for smaller problems the C implementation shows better speed-up, for problems with 450 jobs and more the speed-up of the S-Net implementation is larger than that of the C implementation.

7. RELATED WORK

Stream processing has a long history of research and is well suited for parallel hardware due to the implicit pipeline parallelism that can be easily exploited by stream processing.

An early model of stream-processing have been Kahn Process Networks [13]. A Kahn Process Network consists of processes that read from one or more FIFO input channels and write to one or more FIFO output channels. A process gets fired (i.e., becomes ready) when data are available on all its input channels. Kahn Process Networks are deterministic in the sense that the same input data allways produce the same output data. This is based on the assumption that each process behaves in a deterministic way, i.e., the input data are read in a deterministic order and the process blocks while there are no data available on the input. Though the original specification of Kahn Process Networks defines the capacity of any FIFO channel as infinite, i.e., a write to the channel cannot block. However, real implementations with bounded channel capacity introduce artificial deadlocks.

Stream processing has evolved into a complete programming paradigm, resulting into various approaches combining stream processing with software engineering methods. For example, stream processing has become quite popular for embedded computing with the arising of synchronous strictly time-triggered approaches, like *Giotto* [12], *Scade* [6], or *StreamIt* [25].

Reo is a coordination language for stream processing that deploys software engineering methods to structure streaming networks hierarchically [2]. Connectors are explicit entities in Reo. The atomic connector is called channel and exposes properties like capacity, lossy communication, or allowing only synchronous read and write. Connectors can be constructed as networks of channels and may be reconfigured at runtime. *WaveScript* is an example of a stream-based programming language that does not follow the concept of a coordination language, as stream-based communication and logic programming are interweaved [17].

S-Net positions itself as a stream-based coordination language that supports asynchronous and non-deterministic computation. Further, S-Net facilitates typed messages and provides language interfaces for different process implementation languages.

8. CONCLUSION

The paper focuses on a particular form of combinatorial technique called Ant Colony Optimisation, which is known to be useful for solving various graph-based problems of practical significance. Using an application of ACO as an example of a highly irregular, distributed problem, we have demonstrated that the data-flow-style, stream-processing component technology S-Net facilitates both the development and distribution/parallelisation of the code with high-level and easy to use language constructs, while keeping the performance in the same league as the hand-coded solutions utilised by industry.

The performance evaluation of the ACO implementation shows that the performance of S-Net lies within the same range as that of hand-parallelised C code. Still, as the maximal measured speed-up on a 48-core machine is slightly below 24 further investigations are required to identify ways to increase the efficiency of the S-Net implementation. A finer granularity of the decomposition of the application is expected to be essential in achieving a higher utilisation of the machine. Experiments with additional problem sizes, different scheduling techniques and dynamic load balancing will provide further insight and guidance towards finding current inefficiencies. Also, a measurement framework that allows for capturing several properties such as execution time, waiting time, throughput and latency of single tasks and networks is currently under development. Even in its current prototypical state it already helps to pinpoint hot-spots in the application and the S-Net runtime system. Work in this area is expected to substantially contribute to future developments of the S-Net tool chain. Ultimately, we plan to use runtime observations to apply dynamic network optimisations on the coordination level as well as to provide the compilation process of the box language with collected information in order to generate more specialised and optimised code.

On the algorithmic side, future steps include producing a variety of stream processing schemes for ACO, coded in S-Net, to investigate their relative efficacies.

Acknowledgements

This work is partially supported by the European Union through the STREP project Advance (Asynchronous and Automatic Virtualisation through Performance Analysis to Support Concurrency Engineering, project no. FP7 248828).

9. REFERENCES

[1] T. S. Abdul-Razaq, C. N. Potts, and L. N. V. Wassenhove. A survey of algorithms for the single machine total weighted tardiness scheduling problem. *Discrete Applied Mathematics*, 26:235–253, 1990.

[2] F. Arbab. Reo: A channel-based coordination model for component composition. *Mathematical Structures in Comp. Sci.*, 14(3):329–366, June 2004.

[3] H. A. J. Crauwels, C. N. Potts, and L. N. Van Wassenhove. Local search heuristics for the single machine total weighted tardiness scheduling problem. *Informs Journal On Computing*, 10(3):341–350, 1998.

[4] M. den Besten, T. Stützle, and M. Dorigo. Ant colony optimization for the total weighted tardiness problem. In M. Schoenauer et al., editors, *Parallel Problem Solving from Nature: 6th international conference*, volume 1917 of *LNCS*, pages 611–620, Berlin, September 2000. Springer Verlag.

[5] M. Dorigo and T. Stützle. *Ant Colony Optimization.* Bradford Book, 2004.

[6] F.-X. Dormoy. Scade 6: A model based solution for safety critical software development. In *Proc. 4th ERTS*, Toulouse, France, 2008.

[7] C. Grelck. The essence of synchronisation in asynchronous data flow. In *25th IEEEIPDPS'11, Anchorage, USA*. IEEE Computer Society Press, 2011.

[8] C. Grelck, A. S. (eds):, F. Penczek, C. Grelck, H. Cai, J. Julku, P. Hölzenspies, S. Scholz, and A. Shafarenko. S-Net Language Report 2.0. Technical Report 499, University of Hertfordshire, UK, 2010.

[9] C. Grelck and F. Penczek. Implementation Architecture and Multithreaded Runtime System of S-Net. In S. Scholz and O. Chitil, editors, *Proc. IFL'08*, volume 5836 of *LNCS*. Springer-Verlag, 2010.

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

[11] C. Grelck, S.-B. Scholz, and A. Shafarenko. Coordinating Data Parallel SAC Programs with S-Net. In *Proceedings of IPDPS'07, Long Beach, California, USA*. IEEE Computer Society Press, Los Alamitos, California, USA, 2007.

[12] T. A. Henzinger, C. M. Kirsch, and S. Matic. Composable code generation for distributed Giotto. In *Proc. ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*. ACM Press, 2005.

[13] G. Kahn. The semantics of a simple language for parallel programming. In J. L. Rosenfeld, editor, *Proc. IFIP Congress on Information Processing*, Stockholm, Sweden, Aug. 1974. ISBN: 0-7204-2803-3.

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

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

[16] D. Merkle and M. Middendorf. Fast ant colony optimization on runtime reconfigurable processor arrays. *Genetic Programming and Evolvable Machines*, 3(4):345–361, 2002.

[17] R. Newton, L. Girod, M. C. abd Sam Madden, and G. Morrisett. WaveScript: A case-study in applying a distributed stream-processing language. Technical Report TR-2008-005, MIT/CSAIL, Cambridge, USA, Jan. 2008.

[18] M. Pedemonte, S. Nesmachnow, and H. Cancela. A survey on parallel ant colony optimization. *Applied Soft Computing*, May 2011.

[19] F. Penczek, S. Herhut, C. Grelck, S.-B. Scholz, A. Shafarenko, R. Barrière, and E. Lenormand. Parallel signal processing with S-Net. *Procedia Computer Science*, 1(1):2079 – 2088, 2010. ICCS 2010.

[20] F. Penczek, S. Herhut, S.-B. Scholz, A. Shafarenko, J. Yang, C.-Y. Chen, N. Bagherzadeh, and C. Grelck. Message Driven Programming with S-Net: Methodology and Performance. *ICPP Workshops*, 0:405–412, 2010.

[21] J. C. Phillips, R. Braun, W. Wang, J. Gumbart, E. Tajkhorshid, E. Villa, C. Chipot, R. D. Skeel, L. Kalé, and K. Schulten. Scalable molecular dynamics with namd. *Journal of Computational Chemistry*, 26(16):1781–1802, 2005.

[22] C. N. Potts and L. N. Van Wassenhove. Single machine tardiness sequencing heuristics. *IIE Transactions*, 23(4):346–354, 1991.

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

[24] B. Scheuermann. *Ant Colony Optimization on Runtime Reconfigurable Architectures.* PhD thesis, Universität Karlsruhe (TH), 2005.

[25] B. Thies, M. Karczmarek, and S. Amarasinghe. StreamIt: A language for streaming applications. In *Proc. 11th International Conference on Compiler Construction (CC'02)*, pages 179–196, London, UK, 2002. Springer Verlag.