A Data-Flow Based Coordination Approach to Concurrent Software Engineering

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

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

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

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

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

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

Abstract—**In this paper we present S-Net, a coordination language based on dataflow principles, intended for design of concurrent software. The language is introduced and then used for programming a concurrent solver for a combinatorial optimisation problem. We present the analysis and tracing facilities of our S-Netruntime system and show how these aid programmers in optimising the performance of their applications.**

I. Introduction

Dataflow is a term that is commonly associated with an execution model, where it tends to refer to an instruction scheduling mechanism whereby actions are performed on data when and as they arrive, which is how dataflow differs from control flow, where instructions are scheduled when control is passed over to them, and if the data required is not available yet, then the instructions are delayed accordingly. It is rare to see dataflow appreciated as a software glue, which enables software composition without centralised control and which thus makes software components interact with one another in a scalable manner even within a very large system. Yet there is no reason why the dataflow principle should be confined to just the execution model, rather than the software engineering concept as a whole. The latter at this time is entrenched in its object-centric view, defining as it does object hierarchies and actions taken on those. In a way, designing a large software system nowadays is tantamount to defining and carefully structuring a very large state machine, often with parts of it operating concurrently, thus proliferating state transitions even further. The authors of this paper see the dataflow principle as a glue for combining components into systems whilst avoiding:

- centralisation of control and the cost of frequent nonlocal synchronisation;
- unpredictable blocking due to having to pass control to and from opaque components abstracted behind their interfaces;
- having to make individual components aware of the overall system design by means other than shared namespace;
- insufficient separation between the communication and computation concerns.

The above four-prong challenge lends itself nicely to a dataflow solution at the system level, but if taken all the way through the component levels down to the executing substrate, would necessitate a redesign of the existing application software (and, perhaps, even hardware) base. Even if this were justified for extreme scale computing, one would have to face the fact that medium scale computing would use a different software component base, and, certainly, legacy components would not be directly reusable.

In order to improve software reuse, we propose to employ the second concept, one of *coordination*. Namely, we have developed, implemented and evaluated on some industry-relevant examples a *dataflow coordination language*. In our approach as opposed to codelets [1], the schedulable units of computation are ordinary, imperative procedures, written in a conventional programming language. We do not propose, nor do we depend on any particular concurrency mechanism that supports these procedures. They can use data parallelism inside, or could be CDGs containing codelets under a codelet management system. We can use both kinds of components as well as ordinary sequential code.

The idea of coordination programming is not new, it goes back to the 1980s, when first attempts were made to separate concurrency concerns: spawning parallel threads, communicating between them and synchronising the various activities that form an application – on the one hand, from the computational concerns, which are phrased in terms of programming for some outputs that functionally depend on inputs in a clear and verifiable manner [2], [3] – on the other. Somehow this idea never took off in the mainstream, having been replaced (but not subsumed) by the much less comprehensive idea of middleware [4]. The difference between the former and the latter is similar to the difference between a procedural language and an assembler. The former has (or can have) a sophisticated software engineering layer: a type system, a modularity mechanism and various means for ensuring extensibility and software reuse. The latter is “bare metal”: with a few exceptions the middleware provides a virtualisation level under the control of its client program [5], [6], [7], [8].

That is theory. In practice, all attempts at coordination so far have been oblivious of those software engineering concerns and went conceptually no further than middleware. Compare, for example, MPI for distributed array and task processing (whose many versions are available for clusters, Grids and Cloud) and the claimed coordination language Reo [9], which offers, again, a library of "connectors", to be used with client components. A language as such is more or less absent, just as it was absent in the mother of all coordination languages Linda, whose constructs were little more than invocations of special intrinsic functions.

One would argue though that putting a programming language to the job of coordinating distributed applications might be excessive. After all, the data concept is supposed to be taken care of by the computational part. Why then have a type system? Similarly, encapsulation, inheritance and other aspects of software reuse might be assumed to belong in the domain of the computational language, which the coordination "language" is supposed to extend. Would it not be best to leave the data concept completely opaque to coordination?

Such arguments are fundamentally flawed. Indeed the data concept of the computational language is external to the coordination infrastructure. Nevertheless, hierarchical structures exist inside as well as outside the coordinated components. The topology of the component connections is itself an (abstract) object that can possess all features of a "normal" object: abstraction, encapsulation and inheritance, even without having a state. Certain patterns of communication, such as long pipelines, give rise to additional structuring needs as well: one might see a section of a pipeline as a pipeline in its own right, as well as seeing the process of extending the section by a further stage as something that may require an inheritance mechanism. It is those structuring needs associated with a distributed application topology that make a proper coordination language highly desirable. Also, a fully-fledged coordination language could achieve a much deeper separation of concerns, something that the coordination agenda has been promoting from the start [10].

Topology: The concept of application topology is at the centre of our approach. While original Linda used a completely unstructured tuple-space and while MPI communication is similarly dynamic, we observe that static features of an application's internal connectivity are as important as static knowledge of class structures and data types. Both have the power to enable compiler intelligence, both are useful for optimisation and, last but not least, early detection of programming errors. How is one to handle component connectivity while honouring the engineering concerns identified above?

Conventionally, component networks are conceptualised as directed graphs where the components are placed at the vertices and where the communication channels are represented as arcs. While being a good model for the data streams flowing between the code units, such an approach does not lend itself easily to hierarchical treatment. The reason is the lack of locality in an arbitrary communication graph, similarly to the lack of locality in an arbitrary control graph, something that prompted Dijksstra's famous ”goto considered harmful” all those years ago [11]. The solution to the lack of this locality can be similar to the solution proposed for goto: a network entity should have one input and one output, and if that is too much of a constraint, any multiple inputs or outputs can be tied together, just as gotos were in a nest of control structures. When this idea, hereinafter referred to as the Single-Input-Single-Output (SISO) principle is taken on board fully, it gives rise to almost all features of our approach: the combinators that combine arbitrary SISO entities into compound SISO entities, the type system that provides subtyping and inheritance over SISO structures, the use of nondeterminism in order to be able to merge multiple connections into one and our specific synchronisation solution in the form of a SISO synchrocell.

Dynamic networks: While static structuring results in a great deal of separation between components and the coordination program that controls them, dynamic topologies effect a similar separation between a coordinated program and the distributed platform on which it runs. Indeed, even with the best heuristic agility compilers cannot statically predict the behaviour of a component network when it is mapped on a distributed platform with variable resource utilisation, such as Cloud. Fully automatic adaptation deprives the (coordination) programmer of the means of expressing domain-specific distribution intelligence. When the coordination programmer is able to express dynamic network behaviour: extensions/contractions as more or less resources are needed, this utilises domain specific knowledge of the application. This is not merely an implementation concern since dynamic networks create dynamic hazards for the match between component interfaces. However, here as well, a disciplined approach is possible. We support statically heterogeneous, dynamically homogeneous networks, where expansion is possible but only in the way of replicating something that has been statically defined.

Contribution: This paper gives an introduction to the proposed programming methodology by means of an example application. We begin with section 2, where we introduce our coordination language S-Net developed at the University of Hertfordshire and implemented by a consortium of European universities. Subsequent sections describe the optimisation problem that was brought forward by one of our industrial partners and we present a performance evaluation. Finally there are some conclusions and thoughts for the future.

II. S-Net in a Nutshell

S-Net is a high-level, declarative coordination language based on stream processing and data flow principles. 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, i.e. no information may be carried over between two consecutive box activations. Implementation-wise, S-Net supports (a subset of) ISO-C and the functional array language SAC[12].

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*. Fields are associated with values from the box language domain; they are entirely opaque to S-Net. Tags are associated with integer numbers; they are accessible on both the coordination and box levels.

Following the data flow principle, a box is triggered by receiving a record on its input stream. When this happens, the box applies its function to the record. During execution the box may send records to its output stream. As soons as the function has finished, the box is ready to receive and 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>));
```

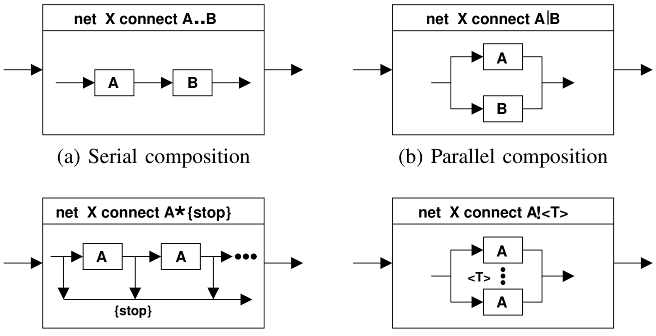
declares a box foo that expects records with a field labelled a and a tag labelled b. Tag labels are distinguished from field labels by angular brackets. The box responds with an unspecified number of records that either have just field c or fields c and d as well as tag e. The associated box function 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 specifying the box interface, we use *sets* of labels in by curly brackets for types. Hence, box foo has type

```
{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*. 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 $w \in y$. S-Net introduces the concept of *flow inheritance*: excess fields and tags from incoming records are attached to any outgoing record produced 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 defines network connectivity as explicit wiring. Instead, it uses algebraic formulae for describing streaming networks. The restriction of the boxes to 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, again is an SISO entity.



(a) Serial composition

(b) Parallel composition

(c) Serial replication

(d) Parallel replication

Fig. 1: Illustration of the four S-Net network combinators

Let A and B denote two S-Net networks or boxes. Serial composition (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.

Parallel composition (A|B) constructs a network where incoming records are either routed to A or to B; their output streams are merged to form the compound output stream. 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 networks' input types match equally well, either alternative is selected non-deterministically.

The serial replication combinator A*type constructs an infinite chain of replicas of box or network 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 A, but this time the replicas are connected in parallel. All incoming records must carry the tag <tag> whose value determines the replica to which the record is routed.

In practice, we often see boxes introduced for housekeeping purposes, such as: renaming, duplication or elimination of fields and tags or simple arithmetic on tag values. Using a fully-fledged box for such tasks is unduly cumbersome. As a convenient alternative, S-Net features built-in *filter boxes* that let us express housekeeping tasks on the level of S-Net. The simplest filter is the empty filter [], that accepts any record and forwards it to the output stream. It serves the definition of bypass routes, as shown in the next section. A complete coverage of filters can be found in [13].

While any box can split a record into parts, we also require means to merge two records into one. For this quintessential synchronisation task S-Net features dedicated *synchrocells*, denoted as `[|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 that matches the other pattern arrives, the two records are merged into one, which is sent to the output stream. Incoming records that only match previously matched patterns are immediately forwarded. This bare metal semantics of synchrocells captures the essential notion of synchronisation in the context of streaming networks. More complex synchronisation behaviours, e.g. continual synchronisation of matching pairs in the input stream, can easily be expressed using synchrocells and network combinators; details can be found in [14]. A full account on S-Net is given in [13], references to other application studies and access to the source code of the entire system may be found at `http://www.snet-home.org`.

III. Programming in S-Net

As already discussed, writing an application using S-Net involves two almost independent stages, the implementation of base-layer functions and the implementation of concurrency management code. The former can be done in a variety of programming languages and even in a manner neutral to hardware or execution platforms. The implementation of the coordination code lies at the other end of the spectrum. The specifics of the application's functionality are irrelevant. The focus is solely on the orchestration of the application's building blocks: to design a coordination scheme for the application that exposes its exploitable concurrency to the underlying machine and its parallel computing resources.

We will illustrate our approach by an example, suggested by the industry authors of this paper and which is motivated by the business of SAP AG: production scheduling by ant-colony optimisation. We will deal with both aspects, functionality and coordination, but will only briefly sketch out the internals of the application (Section III-A) and devote most attention to the data flow concurrency engineering aspects (Section III-B).

A. Ant-Colony Optimisation for Scheduling Problems

We are interested in solving the following scheduling problem: n jobs (items) need to be scheduled on a single machine. Associated with each job j is its processing time p_j, a weight (its importance) w_j and a due date d_j. The goal is to find a job sequence π, i.e. a permutation of the job numbers $(1,\ldots,n)$ that 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 problem, which is commonly known as the "Single Machine Total Weighted Tardiness Problem" or SMTWTP for short, has been shown to be $\mathcal{NP}$-hard [15]. Thus, exact algorithms often fail to calculate the optimal solution in acceptable computation time [16]. Heuristic approaches aim at near-optimal solutions by employing different techniques. Among these, ant-colony based algorithms (ACO) belong to the best performing meta-heuristics [17] for solving SMTWTP.

```
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

We limit ourselves to a bird's eye view of ACO, see [18] for more background information. The typical sequential ACO algorithm is shown in Alg. 1. The algorithm starts with an initialisation step. For SMTWTP this phase reads in n job items, weights and deadlines, sets up a pheromone matrix with initial values and prepares other data structures such as the selection set S that contains all unscheduled jobs, and an empty solution vector to hold the scheduling result in the end. This is followed by an iterative body where m ants repeatedly construct solutions (Line 4) by making a sequence of local decisions. 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. The 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}. For this paper we chose the Apparent Urgency (AU) heuristic [19] to derive these $\eta_{ij} := 1/au_j$. 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 [19]. AU exhibited a competitive performance in prior evaluations [17].

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} reinforces 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. 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^*.

The algorithm executes a number of iterations until a specified stopping criterion has been met, which in our case is a predefined maximum number of iteration steps (Line 2).

B. ACO for SMTWTP in S-Net

There are many ways to design an S-Net. For the application at hand we have opted for a straight-forward solution that may not be optimal in terms of performance, but that gives us the opportunity to step-wise construct a network show-casing many of the S-Net features in an accessible way.

The starting point for the S-Net implementation of the ACO solver for SMTWTP is existing C code that implements a fully functional solver, following the algorithmic structure shown in Alg. 1. We aim to reuse as much of the existing C code as possible, and consequently we follow the function abstractions of the original implementation to determine the set of boxes on the S-Net level. Each step of Alg. 1 is represented by a separate box. In expressing the control flow of the original algorithm in the data-flow programming paradigm of S-Net, the signatures that we define for each box, and consequently for each step of the algorithm, play a vital role.

To get things off the ground we define this set of boxes and their signatures:

```
box Initialize(
  (problem_descr) ->
    (ant_data, <ant_id>, <num_ants>, <max_it>)
    | (best_result, <seen_ants>));

box ConstructSolution(
  (ant_data, <ant_id>) ->
    (ant_data, <ant_id>));

box PickBest(
  (ant_data, best_result,
   <ant_id>, <num_ants>, <seen_ants>) ->
    (best_result, <seen_ants>)
    | (ant_data, best_result, <num_ants>));

box Update(
  (ant_data, best_result, <num_ants>, <max_it>) ->
    (ant_data, <ant_id>, <num_ants>, <max_it>)
    | (best_result, <seen_ants>)
    | (best_result, <finished>));
```

Box Initialize requires the initial problem description, i.e. the number of jobs and their attributes, the amount of ants and the upper bound for the iteration counter. From this input the box produces a data structure ant_data that holds the intermediate data of each ant, such as the current result and its tardiness and a (conceptual) copy of the pheromone matrix. The box outputs this structure for each ant together with a tag <ant_id> that uniquely identifies each ant, the number of total ants <num_ants> and the iteration bound <max_it>. The initialization box also outputs a record to keep track of the overall best result best_result and the number of ants <seen_ants> that have already contributed their result in each iteration. The ConstructSolution box computes a result for one ant as discussed in Sect. III-A and updates the ant_data structure accordingly. Box PickBest inspects the ant_data of each ant and compares it to the current best result best_result. If an ant's result is better than the current best result, best_result is updated accordingly. The box also keeps track of the number of ants that have delivered a result in each round by updating <seen_ants>. After the box has seen all ants in one round it outputs one record containing the overall best result of the current iteration and an updated ant_data structure that is used for the set of ants in the next round. The update box uses the updated ant_data to produce a new set of ants for the next round, if the maximum number of iterations as stored in <max_it> compared to an iteration counter in the ant data has not been reached yet. If the number has not been reached yet, the box also outputs one record containing the best result of the previous round and resets the counter of seen ants <seen_ants> to zero. Otherwise, the box outputs the best result tagged with <finished> to terminate the algorithm. In the latter case the box does of course not produce a new set of ants.

With the box definitions in place it is the next step to build the network that connects the boxes and provides the desired behaviour. For our ACO SMTWTP we will achieve this step-wise by implementing

- concurrent solver instances, one for each ant,
- a merge phase that collects and picks the best result,
- a recursive application of the process until the maximum number of iterations is reached.

For structuring purposes we implement the solving and the merge stage as sub-networks. Sub-networks can be referred to by name in the connect expression of the enclosing network. This provides an opportunity to break down the specification of complex topologies, i.e. large connect expressions, into more maintainable and easily understandable units.

The implementation of concurrently executing ants is possible by using multiple instances of the constructSolution box. Maximal concurrency exploitation is achieved if we allow one instance of the box for every ant. The Initiliaze box already tags every ant with an integer value <ant_id>. We can use this tag in conjunction with the split combinator !:

```
net solve connect (ConstructSolution!<ant_id> | []);
```

Here, the key word net defines a new network named solve as given by the combinator expression following the key word connect. The empty filter [] in parallel to the ConstructSolution box implements a bypass, i.e. [] is a special case of the filter implementing an identity function with no restrictions on the acceptable input types. This is important because Initialize has two output variants, but only the first variant matches the input of ConstructSolution. The bypass, however, lets the second variant flow around the box. We need the second variant in the merge phase once the first results are constructed.

The basic idea for the merge phase is this: Take the current best result and one result produced by an ant, compare and update the best result if necessary and repeat until all results have been inspected. We achieve this by using a combination of a synchrocell, the pickBest box and the star combinator.

```
net merge connect
  ([| {ant_data, <ant_id>},
      {best_result, <seen_ants>} |]
   .. (PickBest | [])) * {ant_data, best_result};
```

Initially, the synchrocell stores the mock best result that is produced by box Initialize and the first result that is produced by one of the ConstructSolution boxes. After the two records have been merged by the synchrocell PickBest processes it and outputs a new record of variant one, i.e. {best_result, <seen_ants>}. As this record does not match the star pattern it travels on to a new instance of the star operand where a new synchrocell stores it. This synchrocell keeps the record until the next result from the solver instances arrives; since the synchrocell in the previous instance of the operand has disappeared no merging takes place and the record does not match the input requirements of the `PickBest` box. The bypass around the box makes sure that the record is forwarded unaltered to the new operand instance and its synchrocell. This process repeats until `PickBest` has seen the results of all ants in which case it outputs its second variant record. This variant matches the pattern of the star and hence the record leaves the network.

The final ingredient to the application is the implementation of the repeated application of the solve and merge stages until the maximum number of iteration is reached. We use a technique that is similar to that of the merge network involving a star pattern that matches only the last variant of the `Update` box. The fully defined network including the `Initialize` box may be written as follows:

```
net aco_smtwtp {
  /* box definitions as above */
  /* networks "solve" and "merge" as above */
} connect
    Initialize
    .. (solve .. merge .. Update) * {<finished>};
```

The operand of the outer star combinator, i.e. `(solve .. merge .. Update)`, is unfolded as long as tag `<finished>` is not present in the output of box `Update`. The first two output variants of the box which do not contain the tag are identical to the two output variants of the `Initialize` box: From the star operand's perspective there is no difference between accepting an input of the initializer box and an input that is in fact the output of the `Update` box in a previous operand instance.

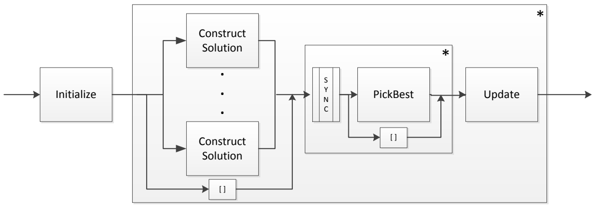


Fig. 2: The ACO SMTWTP network

The network `aco_smtwtp` essentially implements Alg. 1; a graphical representation of the network is shown in Fig. 2. Although this network implements a particular functionality the ideas behind it are quite generic and reusable. In the presented case the network implements a fork-join pattern that we can easily adapt to other applications that can be parallelised using the same pattern.

IV. Performance Evaluation

Analysing the performance of an application is often a complex and time-consuming task. Measuring the total runtime of one execution can easily be done using the `time` command, which is available on many systems, but finding the factors that influence the time requires both thought and effort. Only if we understand how individual parts of a program impact the runtime do we stand a chance to tune the right parameters of the program to improve its overall performance. With concurrent software the problem becomes even more complex, since it is factors such as scheduling and placement, not just the base-layer functions' algorithmic complexity that may have considerable influence on the performance.

The S-Net runtime system is designed with these considerations in mind. It provides us with tracing data that guides the performance analysis and tuning process. The threading backend of S-Net, named LPEL [20], actively manages S-Net components as non-preemptive tasks with low-overhead scheduling to a fixed number of Posix threads for effective utilisation of multi-core processors. The Pthreads are referred to as "worker threads", or just workers. Each worker is statically pinned to a core or CPU of the host machine. If no further configuration parameters are passed to an S-Net executable at startup, LPEL creates one worker for each core and assigns the tasks that are created during the execution of an S-Net to these workers in a round-robin fashion. A distinguishing feature of LPEL is its low-overhead tracing and monitoring subsystem which records execution times of individual tasks, scheduling events and message traces for subsequent performance analysis and optimisations.

In order to properly illustrate these features we shall limit our performance data exploration to comparing the original C implementation, both sequential and multi-threaded, with a few configurations of the data flow, S-Net application.

The measurements presented here are based on execution runs on a 48-core (4 sockets with 12 cores each) AMD Opteron 6174 machine with 256 GB of main memory running Linux with kernel version 2.6.35.11. The compilation of source code is done using `gcc` version 4.5.1 with optimisation level `-O3`.

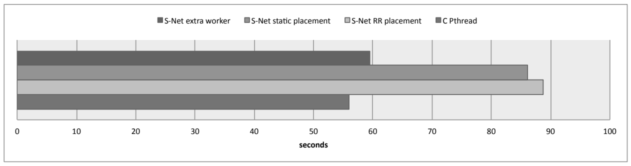


Fig. 3: Runtimes of multi-threaded C and S-Net with different placement strategies

All measurements represent runs of solving the same optimisation problem of 1000 jobs using 45 ants for 500 iterations. Fig. 3 shows the median of the recorded runtimes of 12 runs of the original C implementation of the application which uses Pthreads to implement concurrently working ants. Each ant is mapped onto one thread, i.e. the application uses 45 threads for the ants, and the median of the runtimes is 55.95 seconds with a standard deviation of $\sigma = 0.68$. We compare this to runtimes of the S-Net application as presented in the previous section ("S-Net RR placement" in Fig. 3). The median of the runtimes is 88.8 seconds, $\sigma = 1.14$. Why is the S-Net implementation more than 30 seconds slower than the C implementation? Several factors play a role here: The C implementation works by using globally allocated data-structures that are accessed by all threads concurrently. This allows for an efficient pick-best/update phase after each iteration as all ants have placed their result in one location. The S-Net application implements this phase using a consecutive reduce operation over all results which is carried out sequentially. Also, although the S-Net runtime system was initialised with 45 worker threads the internal placement strategy uses a round-robin assignment of tasks, i.e. box instances, onto these threads as they occur. This means in particular that it is possible for concurrent `ConstructSolution` box instances to be mapped onto the same worker thread if an ant produces a result that is processed by the `PickBest` box before all other ants have been mapped.

As a language for concurrency engineering S-Net allows for user-defined mappings of tasks to worker threads, and in fact to specific cores since worker threads are pinned to individual cores at startup. S-Net provides the `@` combinator for this purpose, which takes an S-Net expression as its left operand and an integer as its right operand. The integer value determines the worker thread that the tasks of the expressions are mapped to. The combinator may also be used in conjunction with the split combinator, as in `!@<n>`, to implement dynamic mapping based on the tag value that is observed by the split combinator (see [21] for details). To see if the above mentioned mapping effects have a negative influence of the runtime we have used a user-defined mapping for the solver instances:

```
net solve connect (ConstructSolution!@<ant_id>|[]);
```

The runtime for this modified network is also shown in Fig. 3 as "S-Net static placement". The median of the runtimes is 86.1 seconds, $\sigma = 0.79$, which is marginally faster than the round-robin version.

In order to further analyse, and ideally improve, the performance of the S-Net implementation we use the monitoring and tracing facilities of the S-Net runtime system [22]. The obtained trace files contain (among other data) information on execution times for individual task instances, grouped by worker threads.

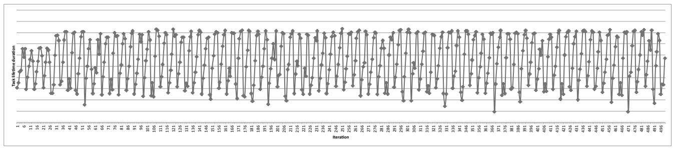


Fig. 4: Lifetime of all `ConstructSolution` tasks on one specific worker

Fig. 4 shows the lifetime of instances of box `ConstructSolution` on one worker thread. The lifetime is measured from the creation of the task until its removal including all suspension times which may occur after reads and writes to and from empty and full buffers. In total we see 500 individual instances of the box, one in each iteration. The lifetimes of the tasks show some periodic fluctuation. This may be explained by the presence of other tasks on the same worker. Since all other tasks of the application are still mapped round-robin to all workers there is periodic competition for compute time between the tasks on a worker. In order to eliminate competition we have also run an experiment in which we use two additional workers for the other tasks of the application:

```
net aco_smtwtp { /* as above */ }
connect
 (Initialize
 .. (solve .. merge@46 .. Update) * {<finished>})@45;
```

This experiment uses 47 worker threads. Threads $0 - 44$ are dedicated to instances of the `ConstructSolution` box, thread 46 runs the merge phase and thread 45 hosts all other tasks. This setup gets us much closer to the runtimes of the original C implementation as can be seen in Fig. 3 "S-Net extra worker". The median of the runtimes for this experiment is 59.45 seconds and $\sigma = 0.89$.

As a final measurement we also compared the multi-threaded C code and the S-Net application using extra workers against optimised sequential C code to get an impression of the scaling behaviour of the different concurrent implementations. We used the same problem size and number of ants for which the median of the runtimes of the sequential C code is 2193.4 seconds, $\sigma = 0.21$. The speed-up of the multi-threaded C code calculates to 39.2, and the speed-up of the S-Net implementation comes to 36.9. Considering that the maximal speed-up lies at 45 for C and 47 for S-Net these numbers leave some room for improvement but we are confident that further analysis and optimisations, especially for the merging phase, will improve these numbers.

V. Related Work

Data-flow and stream processing has been rediscovered and is currently being evolved into a complete programming paradigm, resulting in various approaches that combine data-flow ideas with software engineering methods.

CnC (Concurrent Collections) [23] are based on the same observations as those underlying S-Net, realising that splitting up the development process between a domain expert and a concurrency engineer allows both parties to work more efficiently. Another shared insight is that treating a program as a collection of coarse-grained data-flow blocks provides a very intuitive programming approach that almost naturally leads to a parallel execution strategy.

Similar observations underly TALM and the Couillard compiler [24]. The approach is based on annotated C to mark code blocks for parallel execution and to expose such a block's data dependencies. Being a dedicated language rather than embedded into C source S-Net with its type inference engine allows for more checking and static correctness guarantees, and probably a more high-level approach to engineering a concurrent software project. However, we believe that Couillard is a very interesting compilation target. A combination of the two projects may exploit significant synergy effects, bringing together the efficient execution machinery of Couillard with the high-level semantics analysis of S-Net for consistency checks and optimisations.

Another promising target is the codelet-based execution model proposed in [1]. The approach uses much smaller-grained code blocks than S-Net provides through its box abstraction, however, we see great potential in using the proposed technology to map the task-parallelism exposed by an S-Net program as well as the box-internal data-parallelism onto future chips with much larger core-counts than today, where we can expect the issues of power-efficiency, cost for data movement and task migration to be of major importance.

Worth mentioning are also approaches in the embedded computing domain where various synchronous strictly time-triggered approaches are in use that are based on the same principles, if with a different focus, such as *Giotto* [25], *Scade* [26], or *StreamIt* [27]. *WaveScript* is an example of a stream-based programming language in this domain that does not follow the concept of a coordination language, as stream-based communication and logic programming are interwoven [28].

VI. 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 a data-flow-style, stream-processing component technology called S-Net facilitates both the development and distribution/parallelisation of the code, while keeping the performance in the same league as the hand-coded solutions utilised by industry. The paper has briefly outlined S-Net and has explained its features relevant to the task in hand. It has also presented an analysis of the code performance and discussed the main challenges as well as the extent to which they are met in the current tool chain.

Future steps include producing a variety of stream processing schemes for ACO, coded in S-Net, and investigating their relative efficacies. At the same time the example being reported here is instructive for future S-Net development efforts as a source of usability and performance requirements.

References

[1] S. Zuckerman, J. Suetterlein, R. Knauerhase, and G. R. Gao, "Using a "codelet" program execution model for exascale machines: position paper," in *Proceedings of the 1st International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era*, ser. EXADAPT '11. New York, NY, USA: ACM, 2011, pp. 64–69.

[2] D. Gelernter and N. Carriero, "Coordination languages and their significance," *Commun. ACM*, vol. 35, pp. 97–107, February 1992.

[3] J.-P. Banâtre and D. L. Metayer, "A New Computational Model and its Discipline of Programming," INRIA, Tech. Rep. RR0566, Sep. 1986.

[4] K.-K. Lau and Z. Wang, "Software component models," *IEEE Trans. Softw. Eng.*, vol. 33, pp. 709–724, October 2007.

[5] M. Henning, "A new approach to object-oriented middleware," *Internet Computing, IEEE*, vol. 8, no. 1, pp. 66 – 75, jan-feb 2004.

[6] R. Armstrong, G. Kumfert, L. C. McInnes, S. Parker, B. Allan, M. Sottile, T. Epperly, and T. Dahlgren, "The CCA component model for high-performance scientific computing," *Concurrency and Computation: Practice and Experience*, vol. 18, no. 2, pp. 215–229, 2006.

[7] G. Blair, T. Coupaye, and J.-B. Stefani, "Component-based architecture: the Fractal initiative," *Annals of Telecommunications*, vol. 64, pp. 1–4, 2009, 10.1007/s12243-009-0086-1.

[8] M. Berzins, Q. Meng, J. Schmidt, and J. Sutherland, "DAG-based software frameworks for PDEs," in *Euro-Par 2011: Parallel Processing Workshops*, ser. LNCS, M. Alexander *et al.*, Eds. Springer Berlin / Heidelberg, 2012, vol. 7155, pp. 324–333.

[9] F. Arbab, "Reo: a channel-based coordination model for component composition," *Mathematical. Structures in Comp. Sci.*, vol. 14, no. 3, pp. 329–366, 2004.

[10] A. Omicini and M. Viroli, "Review: coordination models and languages: From parallel computing to self-organisation," *Knowl. Eng. Rev.*, vol. 26, pp. 53–59, 2011.

[11] E. W. Dijkstra, "Letters to the editor: go to statement considered harmful," *Commun. ACM*, vol. 11, no. 3, pp. 147–148, Mar. 1968.

[12] C. Grelck and S.-B. Scholz, "SAC: A functional array language for efficient multithreaded execution," *IJPP*, vol. 34, pp. 383–427, 2006.

[13] F. Penczek, C. Grelck, H. Cai, J. Julku, P. Hölzenspies, S. Scholz, and A. Shafarenko, *S-Net Language Report 2.0*, ser. Technical Report 499, C. Grelck and A. Shafarenko, Eds. Hatfield, UK: University of Hertfordshire, School of Computer Science, 2010.

[14] C. Grelck, "The essence of synchronisation in asynchronous data flow," in *IPDPS'11, Anchorage, USA*. IEEE Computer Society Press, 2011.

[15] J. Lenstra, A. Rinnooy Kan, and B. P., "Complexity of machine scheduling problems," *Annals of Discrete Mathem.*, pp. 343–362, 1977.

[16] 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 J. On Computing*, vol. 10, no. 3, pp. 341–350, 1998.

[17] M. den Besten, T. Stützle, and M. Dorigo, "Ant colony optimization for the total weighted tardiness problem," in *Parallel Problem Solving from Nature: 6th international conference*, ser. LNCS, M. Schoenauer *et al.*, Eds., vol. 1917. Berlin: Springer Verlag, September 2000, pp. 611–620.

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

[19] C. N. Potts and L. N. Van Wassenhove, "Single machine tardiness sequencing heuristics," *IIE Transactions*, vol. 23, pp. 346–354, 1991.

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

[21] C. Grelck, J. Julku, and F. Penczek, "Distributed s-net: Cluster and grid computing without the hassle," in *CCGrid'12, Ottawa, Canada*. IEEE Computer Society, 2012.

[22] V. Nguyen, R. Kirner, and F. Penczek, "A multi-level monitoring framework for stream-based coordination programs," in *12th International Conference on Algorithms and Architectures for Parallel Processing, Fukuoka, Japan*, 2012, to appear.

[23] K. Knobe, "Ease of use with concurrent collections (CnC)," in *Proc. of First USENIX conference on Hot topics in parallelism*, ser. HotPar'09. Berkeley, CA, USA: USENIX Association, 2009, pp. 17–17.

[24] L. A. J. Marzulo, T. A. O. Alves, F. M. G. França, and V. S. Costa, "Couillard: Parallel programming via coarse-grained data-flow compilation," *CoRR*, vol. abs/1109.4925, 2011.

[25] 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.

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

[27] B. Thies, M. Karczmarek, and S. Amarasinghe, "StreamIt: A language for streaming applications," in *Proc. 11th International Conference on Compiler Construction*. London, UK: Springer, 2002, pp. 179–196.

[28] R. Newton, L. Girod, M. C. abd Sam Madden, and G. Morrisett, "WaveScript: A case-study in applying a distributed stream-processing language," Massachusetts Institute of Technology Computer Science and Artificial Intelligence Laboratory, Cambridge, USA, Technical Report MIT-CSAIL-TR-2008-005, Jan. 2008.