Int J Parallel Prog (2014) 42:988–1011
DOI 10.1007/s10766-013-0271-8

An Efficient Scalable Runtime System for Macro Data Flow Processing Using S-NET

Bert Gijsbers · Clemens Grelck

Received: 4 March 2013 / Accepted: 20 August 2013 / Published online: 13 September 2013

Abstract S-NET is a declarative coordination language and component technology aimed at radically facilitating software engineering for modern parallel compute systems by near-complete separation of concerns between application (component) engineering and concurrency orchestration. S-NET builds on the concept of stream processing to structure networks of communicating asynchronous components implemented in a conventional (sequential) language. In this paper we present the design, implementation and evaluation of a new and innovative runtime system for S-NET streaming networks. The FRONT runtime system outperforms the existing implementations of S-NET by orders of magnitude for stress-test benchmarks, significantly reduces runtimes of fully-fledged parallel applications with compute-intensive components and achieves good scalability on our 48-core test system.

Keywords High-level programming models · Declarative parallel programming languages and libraries: semantics and implementation

1 Introduction

The multi-core revolution has brought parallel programming from the niche of high performance computing right into the main stream. As a consequence, programmers who had never thought about parallel processing in their application domains and who received no particular training in these issues are suddenly and rather unexpectedly

B. Gijsbers · C. Grelck (✉)
Institute of Informatics, University of Amsterdam, Science Park 904,
Amsterdam, The Netherlands
e-mail: c.grelck@uva.nl

B. Gijsbers
e-mail: b.gijsbers@uva.nl

exposed to the pitfalls of parallel processing. Conventional parallel programming is considered notoriously difficult. One reason for this is that it intertwines two different aspects of program execution: algorithmic behaviour, i.e. what is to be computed, and organization of concurrent execution, i.e. how a computation is performed on multiple execution units, including the necessary problem decomposition, communication and synchronization requirements.

S-Net [8,15] is a declarative coordination language whose design thoroughly avoids the intertwining of computational and organizational aspects. S-Net achieves a near complete separation of the concern of writing sequential application building blocks (i.e. *application engineering*) from the concern of composing these building blocks to form a parallel application (i.e. *concurrency engineering*). S-Net defines the coordination behaviour of networks of asynchronous, stateless components and their orderly interconnection via typed streams. We deliberately restrict S-Net to coordination aspects and leave the specification of the concrete operational behaviour of basic components, named *boxes*, to conventional programming languages.

An S-Net box is connected to the outside world by two typed streams, a single input stream and a single output stream. The operational behaviour of a box is characterized by a stream transformer function that maps a single data item from the input stream to a (possibly empty) stream of data items on the output stream. In order to facilitate dynamic reconfiguration of networks, a box has no internal state, and any access to external state (e.g. file system, environment variables, etc.) is confined to using the streaming network. This allows us to cheaply migrate boxes between computing resources and even having individual boxes process multiple records concurrently. Boxes execute fully asynchronously: as soon as data is available on the input stream, a box may start computing and producing data on the output stream. S-Net effectively implements a macro data flow model, *macro* because boxes do not normally represent basic operations but rather individually non-trivial computations.

Although we do not target high performance computing applications with S-Net in particular, any user expects that S-Net programs run reasonably efficiently on parallel commodity hardware. In this paper we propose a highly efficient and scalable runtime system for S-Net, named Front. Front implements dynamically evolving S-Net streaming networks on multi-core multi-processor systems. Whereas previous S-Net runtime systems [7] merely served as proofs of concept for the macro data flow approach as such, with Front we combine all our experience gathered in the mean time to design and implement a low-overhead, high-performance runtime system that can achieve performance levels competitive with more machine-oriented parallel programming alternatives.

We show that for micro-benchmarks, that act as stress tests for the runtime system, Front outperforms the existing S-Net implementations by orders of magnitude. With heavier computations inside each box, which we deem representative for real-world S-Net applications, performance improvements are more modest but nonetheless quite significant. This makes Front an important step forward to make high-level parallel programming with S-Net attractive.

The remainder of the paper is organized as follows. Section 2 introduces S-Net in more detail while Sect. 3 sketches out previous implementations. Section 4 explains the design of the novel Front runtime system, and Sect. 5 discusses our experimental

evaluation of FRONT. We discuss some related work in Sect. 6 and draw conclusions in Sect. 7.

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.

2.1 Boxes

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 organized 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 happens, 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 characterized 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 else 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 component layer.

2.2 Type System

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 on the type level. 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 emphasize 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 multi-variant types, e.g. the output type of box `foo`: A multi-variant 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 excess fields. S-Net implements the concept of *flow inheritance*: excess fields and tags from incoming records are attached to any outgoing record produced by a network entity in response to that record. Subtyping and flow inheritance are indispensable when it comes to making boxes, which were designed in isolation, collaborate in a streaming network.

2.3 Specification of Streaming Networks

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 five network combinators. Any combinator preserves the SISO property: any network, regardless of its complexity, is a SISO entity in its own right.

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. Thus, `A` and `B` operate in a pipeline. Figure 1 illustrates this and the other four network combinators explained below.

Parallel composition (`A|B`) constructs a network where incoming records, depending on their type, are either sent to `A` or to `B`, and their output streams are merged to form the composed network's output stream. In S-Net, 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

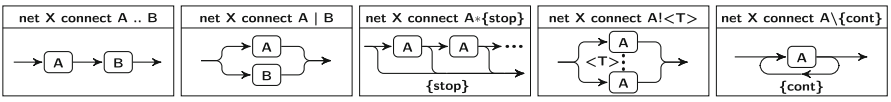


Fig. 1 Illustration of the five S-Net network combinators, which are from *left* to *right*: serial composition, parallel composition, serial replication, parallel replication and feedback

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 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 addition to serial replication S-Net also features a more conventional feedback combinator `A\type`. Here, records always enter subnetwork `A`. If an outgoing record matches the given type pattern, it is sent back to the entry point of `A`; otherwise, it leaves the compound network.

2.4 Filters as Housekeeping Components

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.

2.5 Synchro-Cells—The Essence of Synchronization

While any box can split a record into parts, we also require a means to merge two records into one. For this quintessential synchronization task S-Net features dedicated *synchro-cells*, 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 synchro-cell. 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 synchro-cells captures the essential notion of synchronization in the context of streaming networks. More complex synchronization behaviour, e.g. continuous synchronization of matching pairs in the input stream, can easily be expressed using synchro-cells and network combinators; details can be found in [5].

2.6 Example

Figure 2 shows a simple, yet non-trivial example S-Net coordination program that implements a dynamic graphics filter pipeline; a graphical illustration of the same program is sketched out in Fig. 3.

The top-level pipeline consists of a preprocessing step (`Pre`) transforming an abstract image into its red, green and blue colour components, a dynamic filter pipeline (`Pipe`) and a postprocessing step (`Post`) that turns processed RGB image components back into the original image representation.

The dynamically replicated filter pipeline, implemented with a star-combinator in Fig. 2 and shown in the central compound box in Fig. 3, consists of a splitter that divides an RGB-image (record) into three independent records carrying on the red, green and blue colour information, respectively. These records are routed to three custom filters by means of parallel composition. After the individual processing of colour components, separate red, green and blue records are captured and combined into a single record in the subsequent synchro-cell.

Since we assume a stream of images to be processed by our filter (pipeline) and a synchro-cell only synchronizes a single set of incoming records (see above), we embed the synchro-cell in another serial replication combinator. Consequently, the `Sync` network synchronizes and combines the first red value with the first green and

```
net Example ({Img} -> {Img})
{
  box Pre ( (Img) -> (R,G,B) );
  box fR ( (R) -> (R) );
  box fG ( (G) -> (G) );
  box fB ( (B) -> (B) );
  box Test ( (R,G,B) -> (R,G,B) | (R,G,B,<done>) );
  box Post ( (R,G,B,<done>) -> (Img) );

  net Split
  connect [{R,G,B} -> {R} ; {G} ; {B}];

  net Sync
  connect [|{R},{G},{B}|] * {R,G,B};

  net Pipe
  connect (Split .. (fR | fG | fB) .. Sync .. Test) * {<done>};

} connect Pre .. Pipe .. Post;
```

Fig. 2 Example S-Net implementing a dynamic graphics filter pipeline

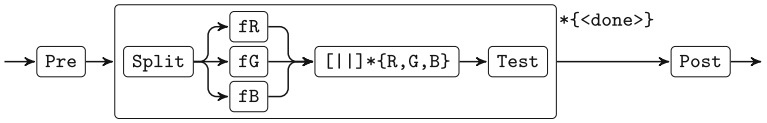


Fig. 3 Illustration of the S-NET streaming network defined in Fig. 2

the first blue value on the inbound stream, the second red with the second green and blue value, and so on. On the combined RGB-image we run a simple test whether to continue filtering or to output the image. The decision is signalled by the presence or absence of the tag `<done>`, which is inspected by the star combinator and later removed in the postprocessing step.

2.7 Summary and Further Reading

To summarize, S-NET is an abstract notation to express concurrency in application programs in an abstract and intuitive way. It avoids the typical annoyances of machine-level concurrent programming. Instead, S-NET borrows the idea of streaming networks of asynchronous, stateless components, which segregates applications into their natural building blocks and exposes the data flow between them. However, S-NET is in no way confined to the area of streaming applications as several case studies successfully demonstrate [9,14,17].

In particular, the concept of asynchronous components connected by streams must not be mistaken for message passing in cluster environments. S-NET as such is architecture-agnostic and both shared and distributed memory implementations have been devised [6]. However, even in a distributed memory scenario today's nodes are highly parallel inside inevitably leading to a 2-level hybrid architecture. Consequently, streams between components in most cases remain within the same memory domain and, thus, do not inflict any overhead with respect to marshalling/unmarshalling of data structures or network transfers.

In essence, S-NET exploits the concept of streaming networks of sequential processes as an intuitive mental model to express concurrency and dependencies while implementations are free to deviate from the model within the limits of the streaming semantics to achieve high performance. As will become clear in the remainder of this paper, the FRONT runtime system aggressively exploits this freedom.

3 Implementing S-NET

Figure 4 illustrates the implementation architecture of S-NET. Going top to bottom, the S-NET compiler takes an S-NET coordination program and compiles it to the S-NET *Common Runtime Interface (CRI)*. This is a well-defined interface that exposes the structure of an S-NET streaming network as an application-specific call tree of application-agnostic library functions instantiated with again application-specific data structures. The library functions of the common runtime interface can be instantiated with alternative implementations and thus allow for entirely different technical realizations of S-NET, as for instance FRONT.

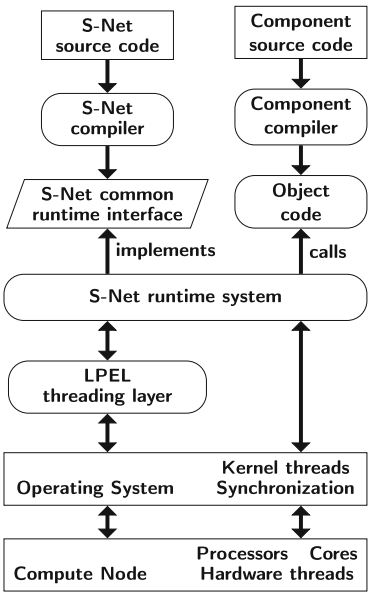


Fig. 4 S-NET implementation architecture

3.1 The Existing S-Net Runtime System

The original runtime system [7] for (shared memory) parallel systems, let us call it S-RTS, sticks closely to the intuition and the operational semantics of S-NET[16]. It sets up a system of communicating sequential processes (CSP). Each S-NET component is instantiated as a sequential process, which executes an event loop that reads a record from the input stream (potentially blocking on an empty stream) and processes that record. In the case of user boxes this usually involves calling an external function implemented in a component language and compiled to binary code by the corresponding component compiler.

During component execution, one or more records may be emitted on the output stream. Termination of the box function completes the event loop and the component is ready to receive and process the next record on the input stream. The process continues until a special input record signals the component to terminate due to global network shutdown or partial network garbage collection. It is noteworthy that records are never copied when moving from one component to another, but we use a reference counting scheme for automatic memory reclamation.

On the S-NET language level stream split and merge points are implicitly represented in the semantics of the parallel composition combinator as well as both replication combinators. In the runtime system explicit *dispatch* and *collect* components take over these routing tasks. Unlike box components, a dispatcher has a single input and multiple output streams while a collector has multiple input streams and a single output stream. Both dispatchers and collectors are implemented by a similar event loop as box components.

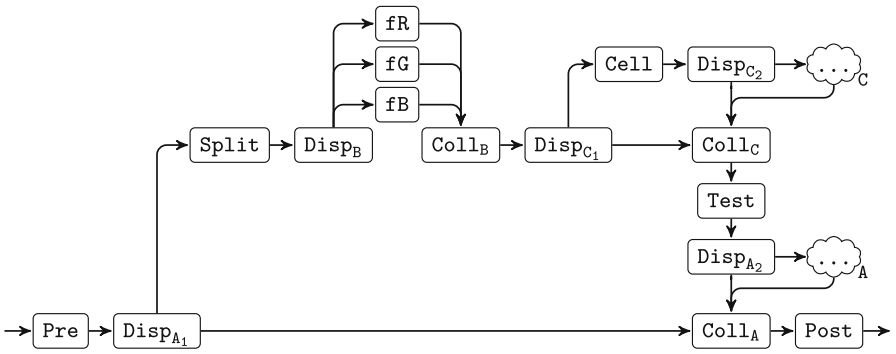


Fig. 5 Illustration of the S-Net runtime system executing the streaming network defined in Fig. 2 and sketched out in Fig. 3; replication combinators are instantiated exactly once; clouds depict potential further instantiations

Both serial and parallel replication combinators describe conceptually unbounded streaming networks. The S-Net runtime system implements them through demand-driven dynamic replication of networks; in Fig. 5 we illustrate this by a single level of instantiation and cloud symbols to represent future re-instantiations. It is the task of the dispatchers implementing serial and parallel replication combinators to identify the need for re-instantiation of subnetworks.

Assuming an unbounded (or at least fairly large) number of input records awaiting processing by an S-Net streaming network, we face the problem that choosing S-Net runtime components with non-empty input streams for execution without any further strategy may easily lead to a situation where many input records are loaded into the network, but very little progress is made towards completing the processing of individual records and emitting them on the overall output stream. Simultaneous in-memory representation of many records may exhaust the available memory. In order to ensure that an S-Net streaming network makes some progress towards completing records we use (fairly small) bounded buffers to implement streams. Accordingly, components may also block on full output buffers. This creates a form of *back pressure* that ensures sufficient progress in practice.

3.2 Mapping Components to Execution Resources

The S-Net runtime system itself is resource-agnostic. There are two alternative scenarios to map S-Net components to the underlying hardware for execution. One maps components one-to-one to kernel threads and leaves their scheduling to the operating system. We refer to this as S- RTS/Pth. The other scenario makes use of the custom-developed *Light-weight Parallel Execution Layer* (LPEL) [18] to explicitly map the dynamic number of S-Net components to a fixed, given number of kernel worker threads. LPEL serves several related purposes. It avoids the creation of a potentially large (since application-driven) number of kernel threads, which is inefficient and may even reach hard limits, and it implements more efficient cooperative scheduling techniques instead of the operating system's preemptive techniques and thus can use

more efficient techniques for synchronization. In the sequel we refer to this variant as S-RTS/LPEL.

4 The FRONT Work Stealing Runtime System for S-NET

Over the past years the existing runtime system of S-NET (S-RTS) has enabled us, and other research groups, to gain extensive and valuable experience with developing a range of different applications using S-NET as a coordination language. This also led to the identification of reasons for observed performance limitations, which we describe here. We formulate new requirements for an improved S-NET runtime system design, which we realize with FRONT.

4.1 Consequences of the Previous Design

To understand the consequences of the design of S-RTS we recall that an executing S-NET program is often highly dynamic due to the presence of serial and parallel network replication ("*" and "!" combinators). As a consequence, record processing frequently cannot progress because required parts of the processing network still await instantiation. We can observe this in Fig. 5 where initially only entities `Pre`, $\texttt{Disp}_{\texttt{A}_1}$, $\texttt{Coll}_{\texttt{A}}$ and `Post` exist. The subnetwork which starts from $\texttt{Disp}_{\texttt{A}_1}$ via `Split` to $\texttt{Disp}_{\texttt{A}_2}$ and ends at $\texttt{Coll}_{\texttt{A}}$, is only instantiated upon arrival of the first record which does *not* carry tag `<done>`. The thread which executes the *star dispatcher entity* $\texttt{Disp}_{\texttt{A}_1}$ then sequentially creates the entities `Split`, $\texttt{Disp}_{\texttt{B}}$, `fR`, `fG`, `fB`, $\texttt{Coll}_{\texttt{B}}$, $\texttt{Disp}_{\texttt{C}_1}$, $\texttt{Coll}_{\texttt{C}}$, `Test` and $\texttt{Disp}_{\texttt{A}_2}$. This requires a significant amount of work. Creating one entity involves:

1. Create a new stream to connect an existing entity with the new entity. Because of blocking semantics, this includes allocating mutex locks and condition variables.
2. Invoke a compiler generated application-specific entity creation function. This creates lists with type configuration and function pointers to subsequent entities.
3. Invoke an entity type specific function to initialize a structure with the received type configuration, create an outgoing stream, and create a thread of execution.
4. The newly created thread blocks when reading from the empty input stream.

For entity $\texttt{Disp}_{\texttt{A}_1}$ the instantiation of the subnetwork results in two streams: one stream connects $\texttt{Disp}_{\texttt{A}_1}$ with entity `Split` and one is to connect $\texttt{Disp}_{\texttt{A}_2}$ with the *collector entity* $\texttt{Coll}_{\texttt{A}}$. At this point in time the latter stream is still unknown to $\texttt{Coll}_{\texttt{A}}$. The single reason why $\texttt{Disp}_{\texttt{A}_1}$ is constructing the entire subnetwork in one go, is that it is now able to communicate this last stream to $\texttt{Coll}_{\texttt{A}}$, by means of a control message, over the stream which directly connects $\texttt{Disp}_{\texttt{A}_1}$ with $\texttt{Coll}_{\texttt{A}}$. Upon reception of this message, $\texttt{Coll}_{\texttt{A}}$ extracts the stream and adds it to its set of incoming streams.

Finally, $\texttt{Disp}_{\texttt{A}_1}$ transmits the record which triggered the subnetwork instantiation, to the input stream of the `Split` entity. We conclude by noting that the construction of the subnetwork is done by a single thread: the thread which executes the $\texttt{Disp}_{\texttt{A}_1}$ entity. Hence, this is serial code which reduces speedup for S-NET applications.

The design of S-RTS follows the process model, where each entity is executed by a dedicated thread of execution. Because S-Net computations unfold in a pipelined fashion, records traverse a succession of entities. This affects performance twofold:

- Potentially at each entity the local thread may have to be woken up and scheduled to a processor core. Context switches of threads are therefore frequent.
- Threads of successive entities likely run on different cores. Therefore, a record may visit many different cores while it traverses the network: data locality is lost.

All this overhead may be acceptable for those S-Net applications where box components require a substantial amount of time to process a single record, but this does not hold for all applications. A new runtime system has to be developed to make S-Net interesting and relevant for applications which require many more entities, or which instantiate entities at a much higher rate.

4.2 New Requirements for Improved Runtime Performance

Based on the above observations, we formulate new requirements for an S-Net runtime system with much improved performance characteristics:

(a) Entity creation must proceed concurrently with record processing as much as possible. Avoid delays which arise due to creating several entities at once. Only create a new entity when it is first needed to process an incoming record.

(b) Creation of one new entity must be cheap and fast. Specifically, to accomplish this avoid creating a new thread of execution for each entity.

(c) Reduce the migration of records over cores when they traverse the network of entities. In order to improve data locality allow for a single thread to continue processing the same records, while they traverse the network.

(d) Evaluate each of the compiler generated application-specific entity creation functions just once. The entity parameters remain unmodified anyway. They can therefore be shared by different instantiations of the same entity. This reduces the memory footprint and enables the reuse of data which is already present in processor caches. This makes entity creation much cheaper and faster.

4.3 Making Entities Small in Size and Fast to Create

Addressing requirement **(d)** one of the key design ideas behind the Front runtime system is to replace the dynamically evolving graph of communicating sequential processes characteristic for the original S-RTS runtime system by two complementary graphs: the static *property graph* and the dynamically evolving *entity graph*.

At program startup execution of the compiler-generated function call graph (see Sect. 3) creates the static *property graph*. In Fig. 6 we show the property graph for our running example. It merely contains placeholders for the replication combinators. The property graph contains all information required for running the network, e.g. types for routing decisions, and serves as a template for evolving the entity graph.

The entity graph facilitates implementation of requirement **(a)**. Initially, Front creates just the first entity, in our example this is `Pre`. Even the creation of the outgoing

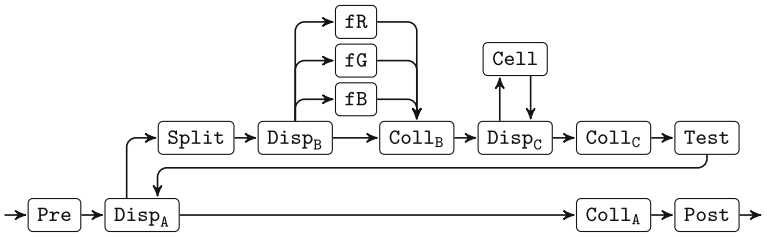


Fig. 6 The static directed graph with node and stream templates, as created by FRONT for the running example introduced in Fig. 2. It serves as a template for instantiating the networks in Fig. 5

stream is postponed. The first record to leave `Pre` will detect the absence of an outgoing stream. The `Pre` entity structure contains a pointer to the `Pre` node in the static node graph of Fig. 6. This suffices to give the outgoing edge, which in turn gives the destination node. From the destination node the entity type is available (in this case a *star dispatcher*) together with application-specific type parameters (e.g. the termination pattern is `<done>`).

In this case the destination is a dispatcher for a star combinator: $\texttt{Disp}_{\texttt{A}_1}$. This requires special handling, because it has two outgoing connections which ultimately both need to end up at the same collector $\texttt{Coll}_{\texttt{A}}$. In order to be able to only allocate a stream when it is first needed to output a record, but still guarantee that all incoming streams to a collector use the same destination entity, each entity carries with it a stack of pointers to future collectors. When $\texttt{Disp}_{\texttt{A}_1}$ is created, the collector $\texttt{Coll}_{\texttt{A}}$ is created as well and pushed onto the stack. Each new entity receives a copy of the stack from its predecessor. When the streams layer detects (according to the static property graph) that it is opening a stream to a collector, then it will take the collector from the top of the stack. We illustrate this with the (simplified) C code in Fig. 7.

4.4 Worker Threads and Work Organization

Requirement **(b)** says to do away with the process-oriented design of one-thread-per-entity, may it be preemptive kernel threads as in S- RTS/PTH or cooperative logical threads as in S-RTS/LPEL. This implies that a single *worker thread* should operate multiple, or potentially all, entities. This has serious consequences for the design of the runtime system. A thread can no longer block when the entity is writing to a full stream buffer, because the executing thread is also needed to process records at other entities. Because a box component is free to write an unspecified number of records to its output stream, the consequence is that output streams of box components (and thus all streams) must be unbounded.

At program startup we create a fixed set of *worker* threads using POSIX threading facilities: one worker per processor core. An *invocation* is the processing of one incoming record at an entity by a worker. To preserve the sequential ordering between records and to guarantee the integrity of an entity, workers must obtain exclusive ownership of an entity before invoking it. This is accomplished by setting a flag in the entity data structure with a compare-and-swap (CAS) instruction, as efficiently supported in hardware by all modern computing machinery.

```
stream_t* OpenStream(edge_t* edge, stream_t* input) {
  stream_t* output = calloc(1, sizeof(stream_t));
  output->edge = edge;
  output->source_entity = input->destination;
  if (edge->dest_node->type == COLLECTOR)
    output->destination = TopOfStack(input->destination);
  else
    output->destination = NewEntity(edge, input);
  return output;
}

entity_t* NewEntity(edge_t* edge, stream_t* input) {
  node_t* node = edge->dest_node;
  entity_t* entity = calloc(1, sizeof(entity_t));
  entity->node = node;
  entity->stack = CopyStack(input->destination);
  if (node->type == DISPATCHER && edge == node->input_edge)
    PushStack(entity->stack, NewEntity(node->coll_edge, input));
  return entity;
}
```

Fig. 7 Opening a stream also creates a new destination entity, except for collectors which were created together with their (first) dispatcher. The second function parameter is a pointer to an existing open stream that ends at the currently executing entity. Both functions use templates from Fig. 6

Now, the question is how do workers find entities with non-empty input streams. Simply roaming the entity graph searching for activated entities would clearly be inefficient. In particular, collector entities would have to keep track of incoming streams and those have to be examined for non-emptiness. Therefore, to avoid this, the unit of work scheduling for FRONT is the non-empty stream itself. Whenever a worker writes to a stream, it remembers this as a *license to read one record* from that stream for a future invocation at the destination entity. It stores this knowledge in a *stream reference structure*, which contains a pointer to the corresponding stream and a counter for the number of read-licenses it has for that stream. When new read-licenses arrive, the worker looks up the stream reference structure in a hash table which is indexed by a pointer to the stream. To exercise a read-license, the worker first attempts to lock the destination entity. If this succeeds, it decrements the number of read-licenses by one, reads one record from the stream and then invokes the entity. If this was the last read-license, it destroys the stream reference structure and its hash table entry.

The way workers organize their collection of stream references determines the order in which they are processed. Therefore, we must first decide upon an appropriate processing order before we design the appropriate data structure. An entity graph can be regarded as a dynamic pipeline with parallel branches, which evolves from an input entity to an output entity. The edges in this graph are the streams which transport the records. In this picture we wish to preferably schedule those non-empty streams, which have the highest probability of quickly producing output. Each output record reduces the memory footprint and also provides the user with results. Another important motivation is to keep the number of non-empty streams as high as possible in order to increase the concurrency which is exposed to all workers. This is best achieved by elongating and widening the entity graph as much as possible.

To achieve these goals, the workers organize their set of stream references in a singly linked list where the tail of the list is closer to the input entity and the head closer to the output entity. When a worker searches for a schedulable stream, it traverses this list starting at the head looking for a stream with a currently unused destination entity. When found, the worker locks the destination entity of the stream, reads one record from the stream and invokes the entity. If the invocation generates new output records which result in a new stream reference structure then these are inserted before the current position in the list. This achieves that streams closer to the output entity are closer to the head of the list of stream references and, therefore, are preferably scheduled.

4.5 Improved Data Locality

To implement requirement **(c)** and improve data locality we add a specific optimization. We extend write operations to streams with an extra Boolean flag which is true only if the write operation is guaranteed to be the last write operation for the current invocation. Each entity implementation can detect whether or not this is the case for each write operation. If the Boolean flag is set and the destination entity of the stream written to is currently not in use then the worker locks the destination entity and immediately continues processing that entity when finishing the current one. One additional efficiency advantage is that the worker does not have to store and retrieve read-licenses to and from stream reference structures. Because most invocations generate precisely one output record, hash table lookup and creation of new stream reference structures is diminished. In other words, the worker thread follows (or "walks") the record(s) through the entity graph. This strategy also improves data locality as the same worker continues to work on the same data.

4.6 Input Control and Work Stealing

When the list of stream reference structures is empty and the worker has not previously encountered the end-of-file condition on the input entity, it tries to obtain exclusive access to the input entity in order to retrieve records from the input parser. This strategy replaces the concept of back pressure through bounded streams in S-RTS to avoid overloading a streaming network with too many incoming records, potentially exhausting the memory capacity of the computing resource. Instead, new work is only admitted to the streaming network if workers are still idle.

FRONT further allows the user to explicitly constrain the number of records input by the number of records already output. For example, the (optional) specification "`input <= 8 + 2 * output`" says that initially up to 8 records can be input, and for every record output another two records may be input. This hypothetical application apparently consumes two input records to produce one output record, and it can work on the production of 4 output records in parallel. The specification then guarantees that memory is always restricted to the working set which is required to operate on at most 8 input records at a time, regardless of how many records have been input or output before.

As a last resort in case that either there is no input on the global input stream or another worker has already locked the input entity, idle workers (*thieves*) turn into thieve mode and examine the list of stream references of other workers (*victims*) for schedulable streams. We store pointers to workers in a global array. Thieves iterate over this array when searching for work. They remember the previously visited victim and continue with the next worker (round-robin). To reduce contention with victims over their stream reference lists at most one thief at a time may visit a victim.

For the same three motivations given previously thieves preferable *steal* read-licenses for schedulable streams which are likely closest to the output. When they find a stream with a lockable entity, they steal half of the number of read-licenses from the victim's stream reference structure. Then they retract from the victim's list and continue with invoking the destination entity for the stolen stream read-licenses. Victims and thieves must exclusively lock a stream reference structure with a CAS (compare-and-swap) instruction before dereferencing a stream pointer or modifying its contents.

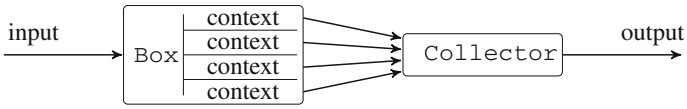
It is noteworthy that our variant of work stealing deviates from the standard approach found in most implementations elsewhere. Normally, work lists are doubly linked, and the owner reads from one end while the thieves read from the other end. This model avoids access conflicts between owners (or victims) and thieves as long as there is more than one work item in the list. In the FRONT runtime system we make use of a single-linked list and both victims and thieves read from the head of the work list. This is necessary to ensure that the S-NET network as a whole makes progress towards producing completed records at the global output. In other words, even in the presence of work stealing the FRONT runtime systems aims at computing tasks at the "front" of the streaming network.

Where a worker looks first for more work when its own work list becomes empty is an important design decision. We choose workers to first check global input for more work before trying to steal work from other workers. This choice reduces the overhead created by many workers simultaneously aiming at stealing work that simply does not exist. Moreover, it helps to accelerate the initial ramp up phase of any S-NET network when the number of records in the system is still small and effectively no work exists that could be stolen. As only one worker at a time can lock and thus operate the input entity, new records may enter the streaming network while at the same time other workers aim at stealing work from their peers.

4.7 Concurrent Invocations of Box Components

The S-NET language specifies box functions to be stateless and functional. As in the FRONT runtime system box components are not associated with an individual thread of control, we can exploit this requirement to significantly increase concurrency of box components by allowing multiple workers to invoke a box entity concurrently as soon as multiple input records are waiting in the input stream. For the purpose of experimentation, a per-box concurrency limit can be specified on program startup. We allocate for a box entity an equivalent number of box contexts. Each box context has its dedicated outgoing stream, which ends at a shared per-box collector. The collector

merges the incoming streams into one outgoing stream. When a worker invokes a box, it finds an unused box context, locks it and if the number of concurrent invocations is below the limit, it immediately unlocks the box entity, to allow for more concurrent invocations by other workers. The current box concurrency level is maintained in an atomic counter. The following figure illustrates a box with the concurrency level set to 4:



The collector entity ensures that despite concurrent box invocations the stream order semantics of S-Net are preserved, i.e. records cannot coincidentally overtake other records.

The ability of the Front runtime system to invoke the same box instance multiple times concurrently if multiple input records are waiting to be processed is an important step to fully exploit the concurrency contained in an S-Net specification. Following the macro data flow approach, the unit of computation in S-Net is the record, not the box component that waits for records. Conversely, in a *communicating sequential process* implementation model, as the original S-Net runtime system does, opportunities for concurrent computations are regularly left out whenever multiple records start queuing in the input stream of a busy box component.

5 Performance Evaluation

We evaluate Front by comparing its performance with that achieved by S-RTS for a variety of applications. More precisely, we compare the following S-Net runtime system configurations:

- S- RTS/Pth: the original runtime system with the Pthread threading layer,
- S-RTS/LPEL: the original runtime system with the LPEL threading layer,
- Front: the novel runtime system introduced in this paper.

Our experimental system is a 48-core SMP machine with 4 AMD Opteron 6172 "Magny-Cours" processors running at 2.1 GHz and 128 GB of DRAM. Each processor core has 64 KB of L1 cache for instructions, 64 KB of L1 cache for data, and 512 KB of L2 cache. Each group of 6 cores shares one L3 cache of 6 MB. The system runs Linux kernel 2.6.18 with Glibc 2.5. We used GCC version 4.4.6 with the -O3 optimization option to compile all C sources.

We first focus on fairly small benchmarks that deliberately stress test the runtime system design and implementation through large numbers of records, frequent expansion of dynamically replicated subnetworks and negligible computations within components. Towards the end of the section we present our findings for two non-trivial (and more representative) S-Net applications. Most S-Net sources are available online as part of the open source S-Net distribution [21]. A plethora of further experimental data can be found in [4].

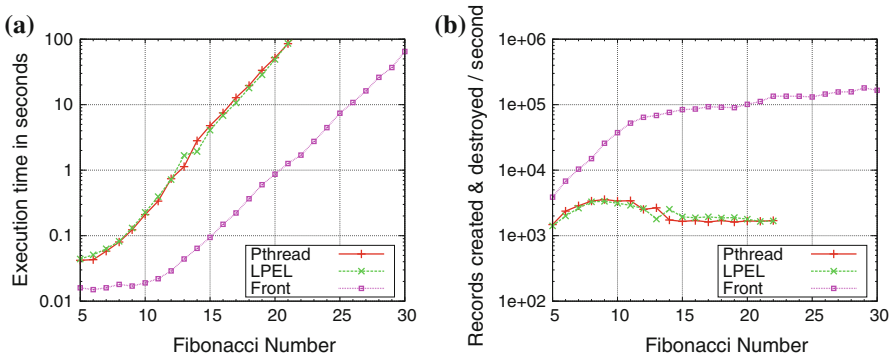


Fig. 8 Performance on the Fibonacci benchmark: **a** execution time, **b** record processing rate

5.1 The Fibonacci Benchmark

In the Fibonacci benchmark[1] we compare the performance of the runtime systems in creating and destroying entities and streams as well as the speed at which records are pushed through the entity graph. Because we omit box components and perform all computations in the coordination language, performance differences by the runtimes are accentuated. We minimize the influence of input parsing and output formatting by accepting only one input parameter and emitting only one output result. The input parameter is the Fibonacci Number which needs to be computed and the single output is its value. We compute the result in a purposely inefficient divide-and-conquer manner such that all S-Net language constructs are used intensively. The number of created records is proportional to the value of the computed Fibonacci Number.

Figure 8a shows that Front is about 50 times faster than S-RTS on this benchmark. Figure 8b uses the same data to show the rate at which records are created and destroyed. For Front this rate increases strongly up to $Fib(12)$ after which it increases weakly, while for S-RTS it diminishes between $Fib(11)$ and $Fib(15)$.

5.2 The PowerOfTwo Benchmark

In S-RTS a collector entity faces the difficult problem of finding among the set of incoming stream connections those streams which are non-empty. This problem is absent in Front because workers only store references to non-empty streams. We demonstrate this with the PowerOfTwo benchmark,[2] which emulates the recursive expansion in S-Net of: `Po2(n){ return (n > 0) ? Po2(n-1) + Po2(n-1) : 1; }`.

We use an index split combinator to create a large number of incoming connections to a collector entity. The maximum number of incoming connections is equal to half the value of the power of two of the input parameter. When we stepwise increase the

[1] Available at https://github.com/snetdev/snet-rts/blob/master/examples/fibonacci/fibonacci3.snet.

[2] Available at https://github.com/snetdev/snet-rts/blob/master/examples/poweroftwo/poweroftwo2.snet.

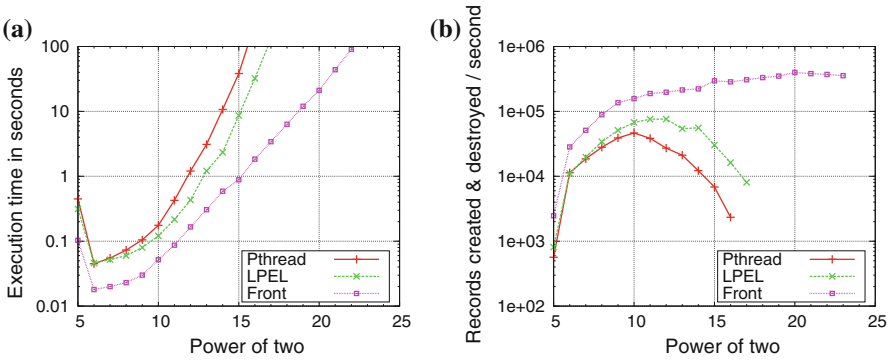


Fig. 9 Performance on the PowerOfTwo benchmark: **a** execution time, **b** record processing rate

```
net StatefulStar ({state} | {inval} -> {outval})
{
    net Combine connect [| {state}, {inval} |];
    box Compute((state, inval) -> (outval) | (state));
}
connect (Combine .. ([] | Compute)) * {outval};

net StatefulFeedback ({state} | {inval} -> {outval})
{
    net Combine connect [| {state}, {inval} |] * {state, inval};
    box Compute((state, inval) -> (outval) | (state));
}
connect (Combine .. Compute) \ {state},{inval};
```

Fig. 10 Two variants of state modelling networks in S-NET

input parameter we reach a point where the collector entity dominates performance negatively for S-RTS.

Figure 9a shows execution times obtained by this benchmark while Fig. 9b shows the corresponding record processing rates. Performance drops significantly for S-RTS when the number of incoming connections increases beyond 1,000. FRONT is unaffected because a worker thread only handles references to non-empty streams.

5.3 State Modeling in the Sieve of Eratosthenes

The Sieve of Eratosthenes is a classic example of a process network [10]. We use it to compare the performance on state modeling in S-NET [5]. Two state modeling design patterns can be expressed in S-NET, as shown in Fig. 10. The first one uses a star combinator over a parallel choice. This network expects one `state` record as input plus a sequence of `inval` records. The box function `Compute` is repeatedly called with two parameters: the current state and a new input value. It is free to decide whether to output an `outval` record, which is the only type of record to leave the network. To continue with evolving the state it must output precisely one `state` record for the next iteration.

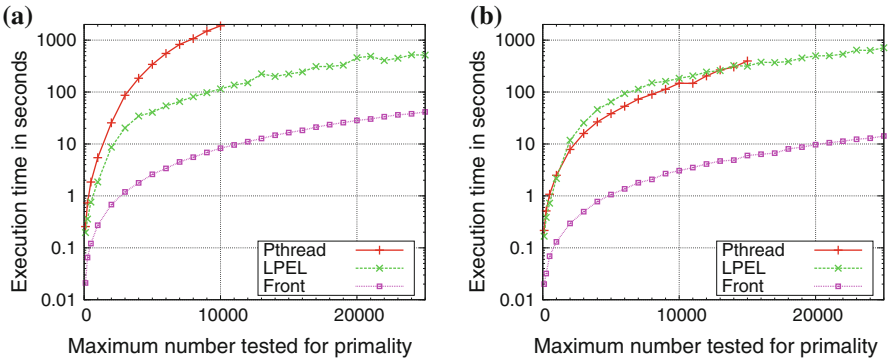


Fig. 11 Performance on State Modeling: **a** star version, **b** feedback version

The second version uses a feedback combinator and avoids the parallel choice. This version uses the common combination of a synchro-cell as operand to a star combinator. Due to the feedback combinator the box `Compute` is instantiated only once. FRONT implements repeated synchronization as a single entity, which allows for considerable efficiency improvements.

We use both design patterns to implement the Sieve of Eratosthenes in S-NET where the found primes are the state records and the inval records are positive numbers to be tested for being prime numbers.[3] Figure 11a, b compare the execution times where the horizontal axis gives the maximum positive number which is tested for primality. The benchmark creates a long pipeline of stateful primality filters. A worker in FRONT may preserve an association with a single record while traversing a number of primality filters and thus avoids the many context switches which occur in S-RTS.

S- RTS/PTH also benefits from the second state modeling design pattern. The reason is that the feedback combinator instantiates its operand at most once. Compared to the first version this significantly reduces the number of created PTHREADs.

5.4 The MTI-STAP Signal Processing Application

MTI-STAP is a signal processing application: Moving Target Indication using Space Time Adaptive Processing [11,17]. Its purpose is to detect slow moving objects on the ground using an airborne radar antenna. The application consists of a long pipeline with 29 different box functions. Some are executed in parallel. We evaluate the performance of this application to see whether the addition of concurrent box invocations to the runtime system can improve performance for existing S-NET applications.

Figure 12a shows execution times for S- RTS/PTH, S-RTS/LPEL and FRONT. Here FRONT runs with the box concurrency limit set to numbers between 1 and 9 as indicated by the suffix: The label FRONT-1 denotes the default configuration, i.e. no concurrent box invocations, while FRONT-9 denotes the configuration when up to nine workers may invoke a single box landing concurrently.

[3] Available at https://github.com/snetdev/snet-rts/blob/master/examples/sieve/sieve.snet.

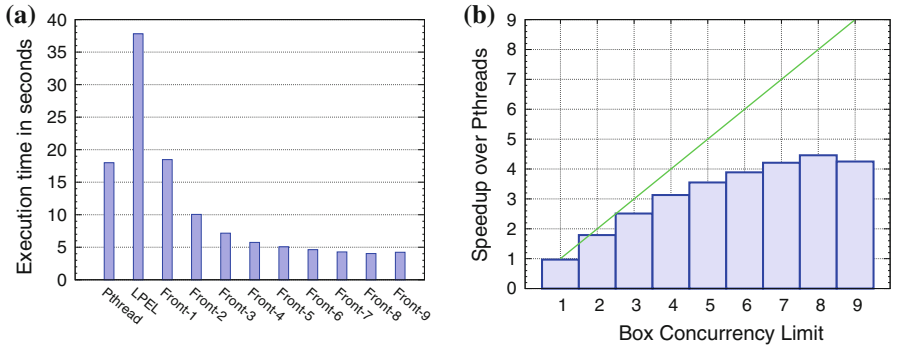


Fig. 12 Performance on MTI-STAP: **a** execution time, **b** speedup for Concurrent Box Invocations

Figure 12b shows the speedup for increasing box concurrency limits relative to the execution time of S- RTS/PTH. This more clearly shows the performance gains by the concurrent box invocations. Of course performance gains by concurrent box invocations are highly application specific. Our implementation merely provides a mechanism for users to increase the exposed concurrency in their applications.

5.5 Speedup on the Cholesky Decomposition Application

Cholesky decomposition is a problem from linear algebra: given a Hermitian positive-definite matrix A, find a lower triangular matrix L, such that $LL^T = A$, where L^T is the transpose of L. We use an implementation[4] by Pāvels Zaičenkovs from University of Hertfordshire, based on the tiled algorithm proposed by Buttari et al [3]. After an initial setup the algorithm repeatedly executes the following seven phases: fan-out, data-parallel computations, fan-in, fan-out, data-parallel computations, fan-in, and sequential consolidation of intermediate results.

We use this application to measure scalability. We stepwise increase the number of processor cores which are available to the runtime system. For this we use the `taskset` program, which permits detailed control of processor core affinity. We incrementally add cores such that they share L3 caches and are part of the same processors and sockets as much as possible. At each measurement step we configure FRONT and S-RTS/LPEL to use a number of worker threads which is equal to the available number of processor cores.

Figure 13a shows measurements where both matrix dimensions are equal to 4,096 and a tile consists of 64 by 64 numbers in 8-byte double precision floating point format. This amounts to 32 KB per tile. Therefore, two tiles fit into the L1 cache of 64 KB. The FRONT runtime system shows a good speedup for 6 cores, diminishing speedup up to 24 cores and no speedup beyond that. The S-RTS runtime shows just a little speedup up to 6 cores, but this disappears when adding more cores, i.e. when more than one L3 cache is involved. Our interpretation of this figure is that the cost to communicate a task to a different core is relatively high compared to the effort required to complete

[4] Available at https://github.com/snetdev/snet-rts/blob/master/examples/cholesky/cholesky.snet.

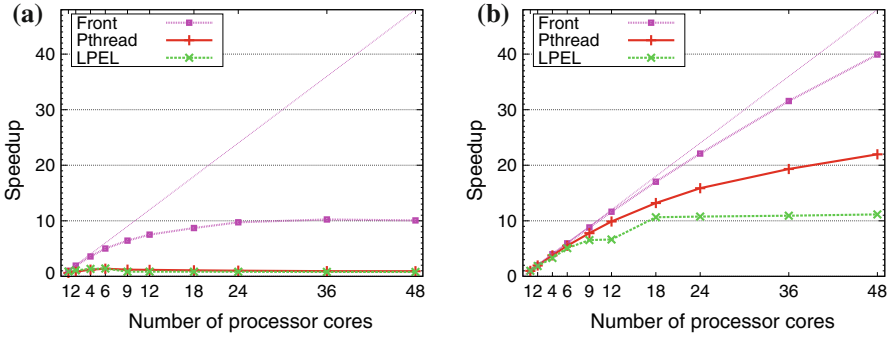


Fig. 13 Speedup on Cholesky decomposition for two parameter sets: **a** matrix size: 4,096 by 4,096, tile size: 64 by 64, **b** matrix size: 8,192 by 8,192, tile size: 128 by 128

the task. S-RTS suffers from the high overhead to construct threads to execute a task, which involves longer sections of serial code and context switches between different threads of execution. In FRONT no new threads need to be created to execute a new task. The construction of entities which facilitate the concurrent execution of a new task only requires some memory allocation, which therefore is much cheaper. Distribution of a task does not require context switches, but at most one work stealing event.

In Fig. 13b we increase the amount of data per tile by four, while keeping the total number of tiles identical. The number of entities which are constructed by the runtime system is the same as well. Only the time spent executing code in the box components increases. Now, S- RTS/PTH also shows reasonable speedup, but less so S-RTS/LPEL. FRONT shows excellent speedup even for 48 cores. Increasing the number of cores from 36 to 48 still improves the performance by 21 %, which we deem satisfactory considering that the algorithm also contains sequential sections.

6 Related Work

Stream processing has a long history and many different interpretations of streams have been used. A survey of stream processing is given by Stephens [20]. A multi-threaded runtime system for the execution of streaming networks can be found for the StreamIt language [22]. Compared to S-NET StreamIt uses synchronous scheduling, while streams and entities in S-NET are scheduled asynchronously.

Work stealing can be traced back to Burton and Sleep who used it to speed up the execution of functional programming languages in a SMP system [2]. Blumofe et al. give a detailed analysis of the time and space complexity for work stealing of tasks with dependencies [1].

Neill and Wierman develop a queueing model of a NUMA system to study work stealing and take into account task affinity. Through simulation of several work stealing and work sharing algorithms under different workloads they find that work stealing insulates from negative effects of affinity and imbalanced load distribution [13].

Mattheis et al. analyze work stealing for streaming applications in soft real-time systems [12]. Here the latency of a task must be kept within bounds. Their model uses

data flow process networks with unidirectional channels. Processes are scheduled when all their input channels are non-empty. Their model supports determinism and multiple concurrent invocations of a single process where input ordering is preserved on output. They compare LIFO and FIFO queueing policies, local queues without or with a shared global queue and two different stealing strategies when a global queue is present. Their model gives bounds on the maximum latency for certain strategies. The best result is found when a global queue is also present which is examined first for stealing before going to local queues of victims.

Sanchez et al. show a scheduler for pipeline-parallel programs which performs fine-grain dynamic load balancing in a runtime system for the GRAMPS programming model [19]. It supports irregular pipeline and data-parallel applications. Their task-stealing uses per-stage queues and queueing policies. Together with back pressure this gives strict bounds on memory usage. Like FRONT, their scheduler also prefers queues which are closer to the output to limit the memory footprint. They do not enforce back pressure on backward queues. Their reasoning is: if the programmer introduces cycles with uncontrolled loops, i.e. loops which produce more data than they consume, then this is incorrect programming, similar to infinite recursion.

7 Conclusions and Future Work

We have presented a novel runtime system for the macro data flow coordination language S-NET, named FRONT. Aiming at highly efficient and scalable parallel execution of S-NET streaming networks, FRONT dispenses with the, at least at first glance, more intuitive interpretation of macro data flow coordination as a growing and shrinking system of communicating sequential (though stateless) components (or processes). Instead, FRONT uses a static property graph to represent network characteristics and a dynamically evolving but information-wise very lean entity graph to trace the dynamic behaviour of a streaming network. The latter is required to ensure the various stream ordering constraints demanded by the S-NET semantics.

A fixed number of worker threads, reflecting properties of the execution resource such as number of cores or hardware threads, roam the entity graph for computational tasks, i.e. components with a non-empty input stream. Upon a match a worker thread executes the task, thus typically creating further tasks. Each worker firstly executes its self-produced tasks to capitalize on data locality in cache-based systems. Only in the absence of further own work does a worker check for new work on the global input stream of the streaming network or try to steal work from other workers. Together with a preference for own tasks that apparently are closer to the global output stream, and thus likely to release (memory) resources soon, these measures successfully avoid overloading of computing resources in streaming scenarios with unbounded data available on the global input stream.

Last not least, FRONT capitalizes on the semantically guaranteed absence of state in S-NET components and runs multiple instances of the same component in parallel if multiple records are available on its input stream. This allows FRONT to harness the full concurrency potential of macro data flow, whereas any solution on the basis of communicating sequential processes would process incoming records in order.

Our experimental evaluation shows that the FRONT runtime system outperforms the existing S-NET implementations by orders of magnitude on benchmarks that stress test the runtime system with components that only perform negligible computations. However, even for real-world applications with compute-intensive box components we can observe considerably improved efficiency, resource utilization, throughput and scalability. Beyond benefitting existing S-NET applications the FRONT runtime system significantly lowers the granularity at which applications can be coordinated efficiently.

Future work includes further evaluation of FRONT on a wider variety of computing architectures and S-NET application programs. Furthermore, we aim at experimenting with implementation variants of FRONT such as different work stealing schemes. For example, we may take the memory hierarchy into account and first attempt to steal work from peers within a L3 cache group. With large numbers of worker threads it is important to avoid having them all trying to steal work simultaneously. For instance, we could allow at most one worker thread per L3 cache group to steal from non-local workers/cores. Another area of future work is in automatically choosing beneficial box concurrency levels (see Sect. 4.7) depending on both application and machine characteristics.

Acknowledgments We thank the anonymous reviewers for their valuable comments that very much helped us to shape the paper. We further thank Pāvels Zaičenkovs for the S-NET implementation of Cholesky factorization.

References

1. Blumofe, R., Leiserson, C.: Scheduling multithreaded computations by work stealing. In: Foundations of Computer Science, 1994 Proceedings, 35th Annual Symposium on, pp. 356–368 (1994)
2. Burton, F.W., Sleep, M.R.: Executing functional programs on a virtual tree of processors. In: Proceedings of the 1981 Conference on Functional Programming Languages and Computer Architecture, FPCA '81, pp. 187–194. ACM, New York, NY, USA (1981)
3. Buttari, A., Langou, J., Kurzak, J., Dongarra, J.: A class of parallel tiled linear algebra algorithms for multicore architectures. Parallel Comput. **35**(1), 38–53 (2009)
4. Gijsbers, B.: An efficient scalable work-stealing runtime system for the S-Net coordination language. Master's thesis, University of Amsterdam, Amsterdam, Netherlands (2013)
5. Grelck, C.: The essence of synchronisation in asynchronous data flow. In: 25th IEEE International Parallel and Distributed Processing Symposium (IPDPS'11), Anchorage, USA. IEEE Computer Society Press (2011)
6. Grelck, C., Julku, J., Penczek, F.: Distributed S-Net: cluster and grid computing without the hassle. In: Cluster, Cloud and Grid Computing (CCGrid'12), 12th IEEE/ACM International Conference, Ottawa, Canada. IEEE Computer Society (2012)
7. Grelck, C., Penczek, F.: Implementation Architecture and Multithreaded Runtime System of S-Net. In: Scholz, S., Chitil, O. (eds.) Implementation and Application of Functional Languages, 20th International Symposium, IFL'08, Hatfield, United Kingdom, Revised Selected Papers, Lecture Notes in Computer Science, vol. 5836, pp. 60–79. Springer (2011)
8. Grelck, C., Scholz, S., Shafarenko, A.: Asynchronous stream processing with S-Net. Int. J. Parallel Program. **38**(1), 38–67 (2010). doi:10.1007/s10766-009-0121-x
9. Grelck, C., Scholz, S.B., Shafarenko, A.: Coordinating data parallel SAC programs with S-Net. In: Proceedings of the 21st IEEE International Parallel and Distributed Processing Symposium (IPDPS'07), Long Beach, California, USA. IEEE Computer Society Press, Los Alamitos, CA, USA (2007)
10. Kahn, G., MacQueen, D.: Coroutines and Networks of Parallel Processes. Technical report, Laboria, University of Edinburgh (1976)

11. Le Chevalier, F., Maria, S.: Stap processing without noise-only reference: requirements and solutions. Radar, 2006. CIE '06. International Conference on pp. 1–4 (2006)
12. Mattheis, S., Schuele, T., Raabe, A., Henties, T., Gleim, U.: Work stealing strategies for parallel stream processing in soft real-time systems. In: Herkersdorf, A., Römer, K., Brinkschulte, U. (eds.) Architecture of Computing Systems: ARCS 2012, Lecture Notes in Computer Science, vol. 7179, pp. 172–183. Springer, Berlin (2012)
13. Neill, D., Wierman, A.: On the Benefits of Work Stealing in Shared-Memory Multiprocessors. Technical report, Department of Computer Science, Carnegie Mellon University (2009)
14. Penczek, F., Cheng, W., Grelck, C., Kirner, R., Scheuermann, B., Shafarenko, A.: A data-flow based coordination approach to concurrent software engineering. In: 2nd Workshop on Data-Flow Execution Models for Extreme Scale Computing (DFM 2012). IEEE Computer Society, Minneapolis, USA (2012)
15. Penczek, F., Grelck, C., Cai, H., Julku, J., Hölzenspies, P., Scholz, S., Shafarenko, A.: S-Net Language Report 2.0. Technical Report 499. School of Computer Science, University of Hertfordshire, Hatfield, England, UK (2010)
16. Penczek, F., Grelck, C., Scholz, S.B.: An operational semantics for S-Net. In: Chapman, B., Desprez, F., Joubert, G., Lichnewsky, A., Peters, F., Priol, T. (eds.) Parallel Computing: From Multicores and GPU's to Petascale, Advances in Parallel Computing, vol. 19, pp. 467–474. IOS Press, Amsterdam, Netherlands (2010). doi:10.3233/978-1-60750-530-3-467
17. Penczek, F., Herhut, S., Grelck, C., Scholz, S.B., Shafarenko, A., Barrière, R., Lenormand, E.: Parallel signal processing with S-Net. In: 10th International Conference on Computational Science, Amsterdam, Netherlands. Procedia Comput. Sci. **1**(1), 2079–2088 (2010)
18. Prokesch, D.: A light-weight parallel execution layer for shared-memory stream processing. Master's thesis, Technical University of Vienna, Vienna, Austria (2011)
19. Sanchez, D., Lo, D., Yoo, R., Sugerman, J., Kozyrakis, C.: Dynamic fine-grain scheduling of pipeline parallelism. In: 20th International Conference on Parallel Architectures and Compilation Techniques (PACT 2011), Galveston Island, TX, USA, pp. 22–32 (2011)
20. Stephens, R.: A survey of stream processing. Acta Inform. **34**(7), 491–541 (1997)
21. S-Net runtime system software distribution (2013). URL:https://github.com/snetdev/snet-rts
22. Thies, W., Karczmarek, M., Amarasinghe, S.: StreamIt: a language for streaming applications. In: Nigel Horspool, R. (ed.) Compiler Construction, 11th International Conference, CC 2002, Grenoble, France. Lecture Notes in Computer Science, vol. 2304, pp. 179–196. Springer, Berlin (2002)