An Efficient Scalable Runtime System for S-Net Dataflow Component Coordination

Clemens Grelck and Bert Gijsbers
University of Amsterdam
{c.grelck,b.gijsbers}@uva.nl

Abstract

S-Net is a declarative component coordination language 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.

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 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 [7, 11] 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 effec-

tively 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 [6] 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.

2 S-Net in a Nutshell

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, but we require any box not to carry over any information between two consecutive activations on the streaming layer. Each box is connected to the rest of the network by two typed streams: one for input and one for output. Messages on these typed streams are 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.

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 labeled `a` and a tag labeled `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.

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.

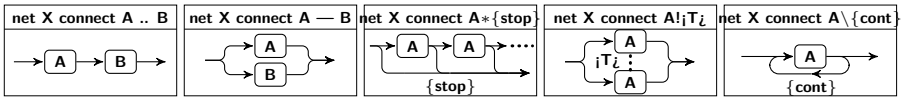

Figure 1: Illustration of the five S-Net network combinators, which are from left to right: serial composition, parallel composition, serial replication, parallel replication and feedback.

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. 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.

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 in series. The chain is tapped before every replica to extract records that match the type specified as the second operand. 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.

In addition to user-defined boxes S-Net features two primitive components: *filters* and *synchro-cells*. Filters mainly serve housekeeping tasks; they can split records, eliminate, duplicate and rename fields, and they support simple computations on (integer) tag values. All the same could be achieved by user-defined boxes, but it is engineering-wise more simple and execution-wise more efficient to have the ability to define such simple tasks on the coordination level as well. Synchro-cells are the one and only means to synchronise independent activities in S-Net. A synchro-cell, denoted as `[|type,type|]`, allows us to merge records of the given types. A record that matches one of the type 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 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].

```
net Example ({Img} -> {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;
```

Figure 2: Example S-Net implementing a dynamic graphics filter pipeline

Fig. 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 the first blue value on the inbound

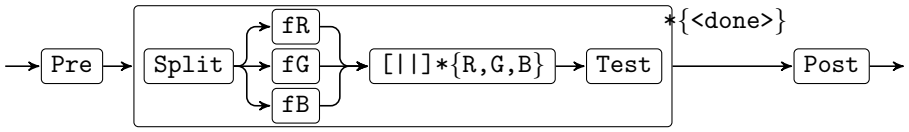

Figure 3: Illustration of the S-Net streaming network defined in Fig. 2

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 signaled by the presence or absence of the tag `<done>`, which is inspected by the star combinator and later removed in the postprocessing step.

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 [8, 13, 10].

3 Implementing S-Net

Fig. 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.

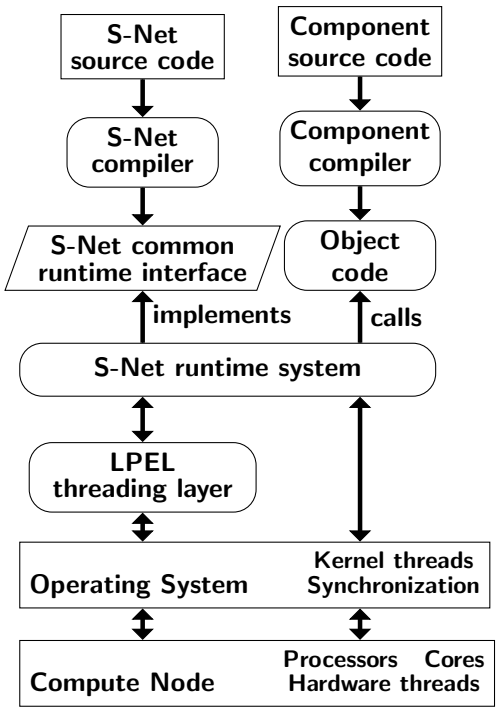

Figure 4: S-Net system architecture

The original S-Net runtime system [6], S-RTS, sticks closely to the intuition and the operational semantics of S-Net[12]. 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. 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. 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 implement the same event loop as boxes.

Both serial and parallel replication combinators describe conceptually unbounded streaming networks. The S-Net runtime system im-

plements 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.

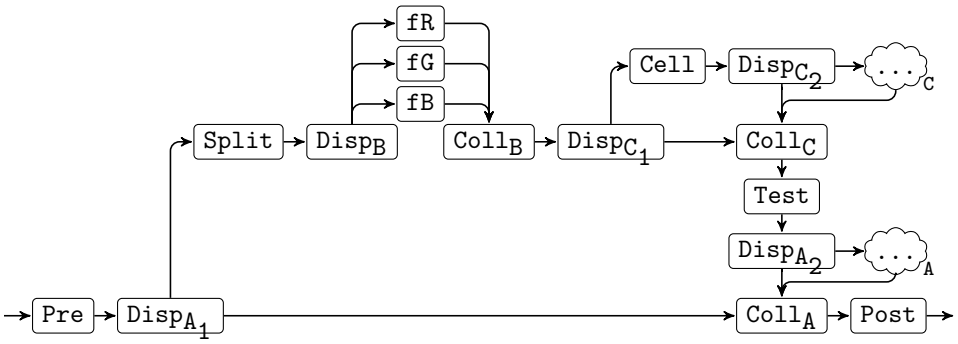

Figure 5: Illustration of the S-Net runtime system executing the streaming network defined in Fig. 2 and sketched out in Fig. 3; replication combinators are instantiated exactly once; clouds depict potential further instantiations

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 tailor-made *Light-weight Parallel Execution Layer* (LPEL) [14] 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 number of kernel threads and it uses efficient cooperative scheduling techniques. In the sequel we refer to this variant as S-RTS/LPEL.

4 The Front Work Stealing Runtime System

The design of the novel Front runtime system is based on an extensive body of experience with the original S-Net runtime system and the LPEL threading layer. S-RTS was mainly intended as a proof-of-concept for macro dataflow computing in the style of S-Net. The addition of LPEL was motivated by supporting meaningful profiling of S-Net streaming networks. With Front we now explicitly aim at scalable performance on large-scale (shared memory) compute servers.

4.1 Entity Graph vs Property Graph

Due to the presence of serial and parallel replication combinators in S-Net, the graph of communicating sequential processes is continuously evolving. It grows due to the demand-driven re-instantiation of argument networks, and it shrinks due to network garbage collection under certain circumstances [5]. Any change in the graph must be attributed to overhead

that competes for resources with productive box computations. In many situations, evolving the network graph additionally reduces the effectively exploitable concurrency. One of the key design ideas behind the FRONT runtime system is, thus, to replace the dynamically evolving graph of communicating sequential processes by two complementary graphs: the static *property graph* and the dynamically evolving *entity graph*.

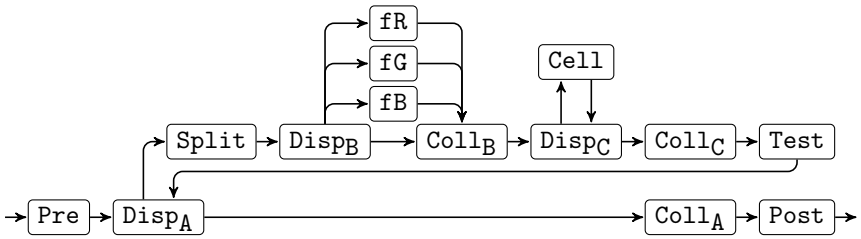

Figure 6: The static entity graph of the running example introduced in Fig. 2.

At program startup execution of the compiler-generated function call graph (the common runtime interface, see Section 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 is created strictly demand-driven. Initially, FRONT creates just the first entity, in our example this is `Pre`. Even the creation of the outgoing 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>`).

Why is it not possible to do with only a static component graph at runtime? The answer is simple: while boxes are stateless by design, whole networks may indeed be stateful as they may contain synchro-cells. At a synchro-cell it is important to match the right records according to the *semantics* of S-NET, which is based on actual replication.

4.2 Process-centric vs Data-centric View

The process-centric view characteristic for S-RTS implementing each entity by a dedicated thread of control, even if implemented as a logical, cooperative thread in LPEL, turned out to be expensive. Thus we aim at a radical change in the interpretation of streaming networks and abandon the process-oriented design. Instead, we create a fixed set of *worker threads* to be run on different cores at application startup. These workers roam the entity graph in search for work. When a worker finds an entity with a non-empty input stream, it locks the entity using a compare-and-swap (CAS) instruction while it processes that entity. One serious consequence of this design is that a thread can no longer block when the entity writes to an already full stream buffer. As the semantics of S-NET does not bound the number of records a box may emit, streams connecting entities must be unbounded, as well.

The question remains how workers find entities with non-empty input streams. Effectively roaming the entity graph would clearly be inefficient. Therefore, the unit of work scheduling in 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. i

4.3 Execution Order

The way workers organize their collection of stream references determines the order in which they are processed. 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.

FRONT stores stream references in a singly linked list per worker; 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 from 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. As a consequence, streams closer to the output entity are closer to the head of the list and, therefore, are prioritised.

4.4 Improved Data Locality

We further aim at avoiding the migration of records from core to core to improve data locality in the ubiquitous presence of hierarchical caches. We extend write operations to streams with a flag that identifies the last output record of some entity invocation. In this case the worker thread immediately continues with the follow-up entity and the current record, i.e. it follows the record through the entity graph. This data-centric (instead of entity-centric) solution has the added benefit that we save storing and retrieving read-licenses.

4.5 Input Control and Work Stealing

When the list of stream reference structures is empty a worker 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. Instead, new work is only admitted to the streaming network if workers are still idle.

If there is neither input on the global input stream or another worker has already locked the input entity, idle workers turn into thieve mode and examine the list of stream references of other workers. 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.

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.

4.6 Concurrent Box Invocation

The S-NET language specifies box functions to be stateless. We can exploit this property to significantly increase concurrency 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 for now. 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 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. 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 in comparison with the original S-Net runtime system with and without the LPEL threading layer. Our experimental system is a 48-core SMP machine with 4 AMD Opteron 6172 "Magny-Cours" processors running at 2.1 GHz and a total of 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 first focus on fairly small benchmarks that stress test the runtime system design and implementation through large numbers of records, frequent expansion of dynamically replicated subnetworks and negligible computation within components before we tend towards real-world applications. Most sources are available online as part of the open source S-Net distribution [1]; more results can be found in [3].

5.1 Fibonacci Benchmark

With the Fibonacci benchmark 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. We do all computing with S-Net filters and minimize the influence of input parsing and output formatting by taking only one input record and producing a single output result, i.e. the argument and result of the Fibonacci function. Our implementation follows a divide-and-conquer approach 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. Fig. 7a shows that Front is about 50 times faster than S-RTS for this benchmark; Fig. 7b shows the (connected) rate at which records are created and destroyed.

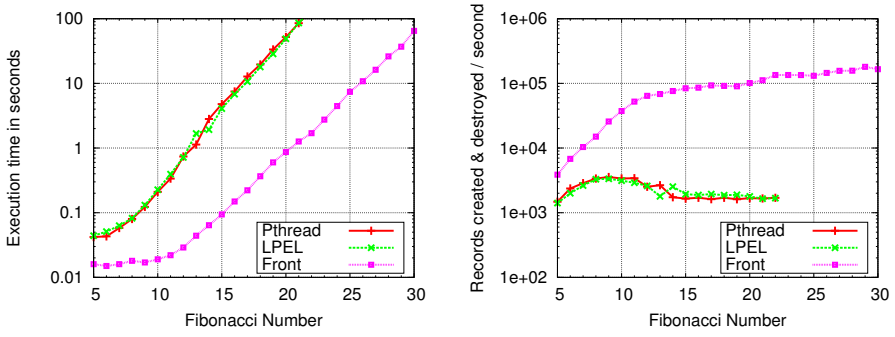

Figure 7: Fibonacci benchmark: **(a)** Execution time, **(b)** Record processing rate.

For FRONT this rate increases strongly up to $Fib(12)$ after which it increases weakly, whereas for S-RTS it diminishes between $Fib(11)$ and $Fib(15)$, regardless of the threading layer.

5.2 PowerOfTwo Benchmark

In S-RTS a collector entity faces the non-trivial problem of finding a non-empty stream among all incoming streams. FRONT solves this issue by only storing references to non-empty streams in the first place. We demonstrate the beneficial effect of this design choice with the PowerOfTwo benchmark, which emulates the recursive expansion of:

```
Po2 n = if n>0 then Po2 (n-1) + Po2 (n-2) else 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 input parameter we reach a point where the collector entity dominates performance for S-RTS, regardless of the threading layer. Fig. 8a shows execution times, Fig. 8b the corresponding record processing rates. Performance drops significantly for S-RTS when the number of incoming connections increases beyond 1,000, whereas scalability of FRONT is unaffected.

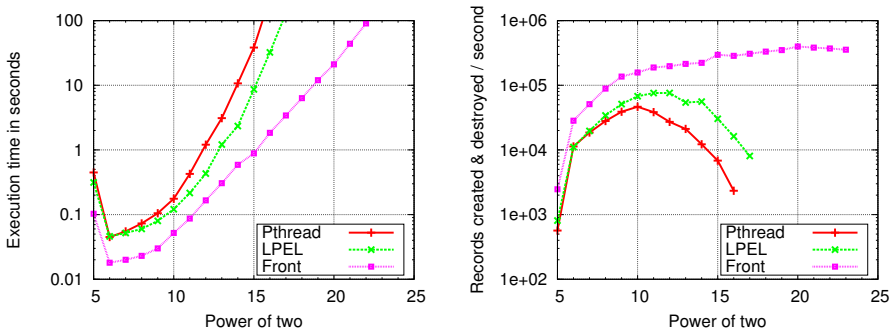

Figure 8: PowerOfTwo benchmark: **(a)** Execution time, **(b)** Record processing rate.

5.3 MTI-STAP Signal Processing

MTI-STAP is a signal processing application: Moving Target Indication using Space Time Adaptive Processing [9, 13]; it detects slow moving objects on the ground using an airborne radar antenna. We evaluate the performance of this application to see if concurrent box invocations in the runtime system can improve performance for existing S-NET applications.

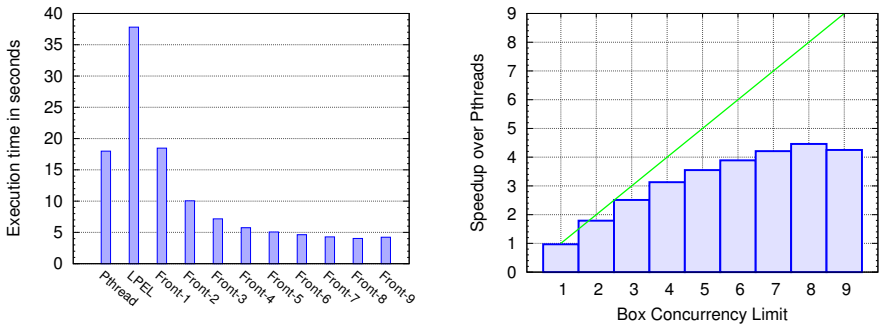

Figure 9: MTI-STAP: **(a)** Execution time, **(b)** Speedup for Concurrent Box Invocations.

Fig. 9a 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. Fig. 9b 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.

5.4 Cholesky Decomposition

Cholesky decomposition is a linear algebra problem: 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 by Pāvels Zaičenkovs, University of Hertfordshire, based on the tiled algorithm proposed by Buttari et al [2]. We use this application to measure scalability and stepwise increase the number of cores. We incrementally add cores such that they share L3 caches and are part of the same processors and sockets as much as possible.

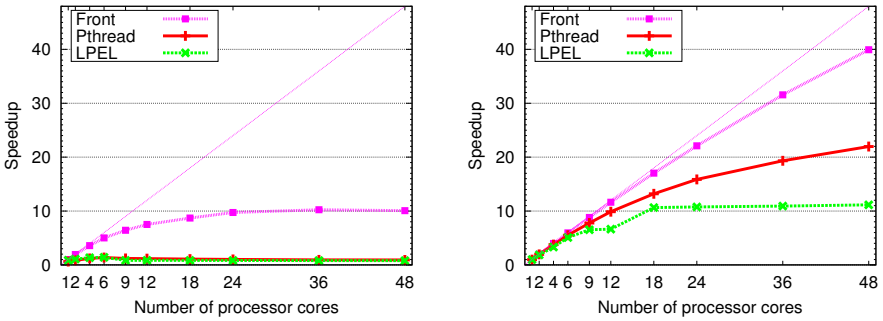

Figure 10: Speedup on Cholesky decomposition for two parameter sets: **(a)** Matrix size: 4096 by 4096, tile size: 64 by 64. **(b)** Matrix size: 8192 by 8192, tile size: 128 by 128.

Fig. 10a shows measurements 4096 by 4096 double precision floating point matrices using 64 by 64 tiles. This amounts to 32 KB per tile. Hence, two tiles fit into the L1 cache of 64 KB. The FRONT runtime system shows good speedups for 6 cores and diminishing speedups up to 24 cores. The S-RTS runtime shows just minor speedups up to 6 cores and no speedups beyond. In Fig. 10b we increase the amount of data per tile by four while keeping the total number of tiles identical. Hence, the number of S-NET entities remains the same as well; only the time spent in box components increases. Now, S-RTS/PTH also shows reasonable speedups, 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 percent, which we deem satisfactory considering that the algorithm also contains sequential sections.

6 Conclusions

We have presented the design, implementation and evaluation of the novel FRONT runtime system for the macro data flow coordination language S-NET. Aiming at highly efficient and scalable parallel execution FRONT dispenses with the intuitive interpretation of macro data flow coordination as a growing and shrinking system of sequential components communicating via bounded streams. Experimental evaluation shows that the FRONT runtime system outperforms the existing S-NET implementations by orders of magnitude for synthetic stress test benchmarks and considerably improves efficiency, resource utilization, throughput and scalability for real-world applications with compute-intensive box components.

Acknowledgements

This paper is an excerpt of a longer article presented at the 7th International Symposium on High-Level Parallel Programming (HLPP 2013) in Paris, France, to appear in the International Journal of Parallel Programming [4].

References

[1] S-Net system software distribution (2013). URL https://github.com/snetdev/snet-rts

[2] 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)

[3] Gijsbers, B.: An efficient scalable work-stealing runtime system for the s-net coordination language. Master's thesis, University of Amsterdam, Amsterdam, Netherlands (2013)

[4] Gijsbers, B., Grelck, C.: An efficient scalable runtime system for macro data flow processing using s-net. International Journal of Parallel Programming (2014). 7th International Symposium of High-Level Parallel Programming (HLPP'13), Paris, France, to appear

[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., Penczek, F.: Implementation Architecture and Multithreaded Runtime System of S-Net. In: S. Scholz, O. Chitil (eds.) Implementation and Application of Functional Languages, 20th International Symposium, IFL'08, Hatfield, UK, Revised Selected Papers, *Lecture Notes in Computer Science*, vol. 5836, pp. 60–79. Springer (2011)

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

[8] Grelck, C., Scholz, S.B., Shafarenko, A.: Coordinating Data Parallel SAC Programs with S-Net. In: 21st IEEE International Parallel and Distributed Processing Symposium (IPDPS'07), Long Beach, USA. IEEE Computer Society (2007)

[9] 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)

[10] 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), Minneapolis, USA. IEEE (2012)

[11] Grelck C., Shafarenko A. (eds.): Penczek, F., Grelck, C., Cai, H., Julku, J., Hölzenspies, P., Scholz, S., Shafarenko, A, Gijsbers B..: S-Net Language Report 2.1. (2013)

[12] Penczek, F., Grelck, C., Scholz, S.B.: An Operational Semantics for S-Net. In: B. Chapman, F. Desprez, et.al. (eds.) Parallel Computing: From Multicores and GPU's to Petascale, *Advances in Parallel Computing*, vol. 19, pp. 467–474. IOS Press (2010).

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

[14] Prokesch, D.: A light-weight parallel execution layer for shared-memory stream processing. Master's thesis, Technical University of Vienna, Vienna, Austria (2011)