Distributed S-Net

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

Jukka Julku
VTT
Technical Research Center of Finland
Espoo, Finland
Email: jukka.julku@vtt.fi

Frank Penczek
Science and Technology Research Institute
University of Hertfordshire
Hatfield, United Kingdom
Email: f.penczek@herts.ac.uk

***Abstract*—S-NET is a declarative coordination language and component technology primarily aimed at modern multi-core/many-core chip architectures. It builds on the concept of stream processing to structure dynamically evolving networks of communicating asynchronous components, which themselves are implemented using a conventional language suitable for the application domain.**

We sketch out the design and implementation of Distributed S-NET, a conservative extension of S-NET aimed at distributed memory architectures ranging from many-core chip architectures with hierarchical memory organisations to more traditional clusters of workstations and supercomputers. Three case studies illustrate how to use Distributed S-NET to implement different models of parallel execution, i.e. pipelined signal processing, client-server and domain decomposition. Runtimes obtained on a workstation cluster demonstrate how Distributed S-NET allows programmers with little or no background in parallel programming to make effective use of distributed memory architectures with minimal programming effort.

***Keywords*-stream processing, component coordination, cluster computing, message passing**

I. Introduction

The historic end of clock frequency scaling and today's hardware trend towards multi-core/many-core chip architectures has brought parallel programming issues from the niche of computational science applications into the main stream of computing. Whereas today's commodity processors from Intel or AMD with four to eight cores are bound to a shared memory model, it is somewhat clear that a substantial increase in core numbers, as envisioned by the manufacturers, can only achieve scalability with a hierarchy of (distributed) memories. Current generations of GPGPU accelerator cards or Intel's new 48-core single chip cloud computer (SCC) already illustrate this likely future trend. Many-core architectures like SCC need to be programmed in one way or another via message passing. And, as soon as multiple machines are to be used cooperatively as a cluster of workstations, there is little hope to avoid message passing at all.

Message passing as a programming paradigm has been studied at least for two decades, and even its most prominent implementation MPI has been around for quite a while. What is new today is the fact that with future chip architectures the message passing paradigm will need to be applied to far less regular problems than in the past, where well structured numerical applications with a regular domain decomposition and communication pattern prevailed. In our opinion this requires a new interpretation of the message passing paradigm that raises the level of abstraction in programming and reasoning such that the challenges of irregular problems can successfully be met.

S-NET [1], [4] is such a novel technology: a declarative coordination language and component technology. The design of S-NET is built on separation of concerns as the key design principle. An *application engineer* uses domain-specific knowledge to provide application building blocks of suitable granularity in the form of (rather conventional) functions that map inputs into outputs. In a complementary way, a *concurrency engineer* uses his expert knowledge on target architectures and concurrency in general to orchestrate the (sequential) building blocks into a parallel application. While the job of a concurrency engineer does require extrinsic information on the qualitative and the quantitative behaviour of components, it completely abstracts from (intrinsic) implementation concerns.

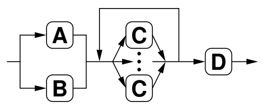


Figure 1. S-NET streaming network of asynchronous components

In fact, S-NET turns regular functions/procedures implemented in a conventional language into asynchronous, state-less components communicating via uni-directional streams. Fig. 1 shows an intuitive example of an S-NET streaming network. The choice of a component language solely depends on the application domain of the components itself. In principle, any conventional programming language can be used, but for the time being we provide interface implementations for the functional array language SAC [2] and for a subset of ANSI C.

Distributed S-NET is a careful extension of S-Net that allows programmers to map sections of streaming networks onto nodes of a distributed memory compute environment with extremely little additional programming effort.

II. S-NET IN A NUTSHELL

As a pure coordination language S-NET relies on a separate component language to describe computations. Such components are named *boxes* in S-NET, their implementation language *box language*. Any box is connected to the rest of the network by two typed streams: an input stream and an output stream. Concurrency concerns like synchronisation and routing that immediately become evident if a box had multiple input streams or multiple output streams, respectively, are kept away from boxes.

Messages on typed streams are organised as non-recursive records, i.e. sets of label-value pairs. Labels are subdivided into *fields* and *tags*. Fields are associated with values from the box language domain. They are entirely opaque to S-NET. Tags are associated with integer numbers that are accessible both on the S-NET and on the box language level. Tag labels are distinguished from field labels by angular brackets. On the S-NET level, the behaviour of a box is declared by a *type 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 that expects records with a field labelled `a` and a tag labelled `b`. The box responds with a number of records that either have just a field `c` or fields `c` and `d` as well as tag `e`. Both the number of output records and the choice of variants are at the discretion of the box implementation alone. The use of curly brackets to define record types emphasises their character as *sets* of label-value pairs.

As soon as a record is available on the input stream, a box consumes that record, applies its box function to the record and emits records on its output stream as determined by the computation. The mapping of an input record to a (potentially empty) stream of output records is stateless. We exploit this property for cheap relocation and re-instantiation of boxes; it distinguishes S-NET from most existing component technologies.

In fact, the above type signature makes box `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 nicely to multivariant types, e.g. the output type of box `foo`: A multivariant type x is a subtype of y if every variant $v \in x$ is a subtype of some variant $w \in y$.

Subtyping on the input type of a box means that a box may receive input records that contain more fields and tags than the box is supposed to process. Such fields and tags are retrieved from the record before the box starts processing and are added to each record emitted by the box in response to this input record, unless the output record already contains a field or tag of the same name. We call this behaviour *flow inheritance*. In conjunction, record subtyping and flow inheritance prove to be indispensable when it comes to making boxes that were developed in isolation to cooperate with each other in a streaming network.

It is a distinguishing feature of S-NET that we do not explicitly introduce streams as objects. Instead, we use algebraic formulae to define the connectivity of boxes. The restriction of boxes to a single input and a single output stream (SISO) is essential for this. S-NET supports four network construction principles: static serial/parallel composition of two networks and dynamic serial/parallel replication of a single network. We build S-NET on these construction principles because they are pairwise orthogonal, each represents a fundamental principle of composition beyond the concrete application to streaming networks (i.e. serialisation, branching, recursion, indexing), and they naturally express the prevailing models of parallelism (i.e. task parallelism, pipeline parallelism, data parallelism). We believe that these four principles are sufficient to construct many useful streaming networks. The four network construction principles are embodied by *network combinators*. They all preserve the SISO property: any network, regardless of its complexity, again is a SISO component.

Let `A` and `B` denote two S-NET networks or boxes. Serial composition (denoted `A..B`) constructs a new network where the output stream of `A` becomes the input stream of `B` while the input stream of `A` and the output stream of `B` become the input and output streams of the compound network. Instances of `A` and `B` operate asynchronously in a pipelined fashion. Parallel composition (denoted `(A|B)`) constructs a network where all incoming records are either sent to `A` or to `B` and their output record streams are merged to form the overall output stream of the network. Type inference associates each operand network with a type signature similar to the annotated type signatures of boxes. Any incoming record is directed towards the operand network whose input type better matches the type of the record itself.

Serial replication (denoted `A*`*`type`*) constructs an unbounded chain of serially composed instances of `A` with *exit pattern* *`type`*. At the input stream of each instance of `A`, we compare the type of an incoming record (i.e. the set of labels) with *`type`*. If the record's type is a subtype of the specified type (we say, it matches the exit pattern), the record is routed to the compound output stream, otherwise into this instance of `A`. Fig. 1 illustrates serial replication as a feedback loop. However, serial replication precisely means the repeated instantiation of the operand network `A` and, thus, defines a streaming network that evolves over time (though in a controlled and restricted way) depending on the data processed.

Indexed parallel replication (denoted `A!<`*`tag`*`>`) replicates instances of `A` in parallel. Unlike in static parallel composition we do not base routing on types and the best-match rule, but on a tag specified as right operand of the combinator. All incoming records must feature this tag; its value determines the instance of the left operand the record is sent to. Output records are non-deterministically merged into a single output stream similar to parallel composition.

To summarise we can express the S-NET sketched out in Fig. 1 by the following expression:

```
(A|B) .. (C!<t>)*{p} .. D
```

assuming previous definitions of A, B, C and D. The choice of network combinators was inspired by Broy's and Stefanescu's network algebra [3].

While any box can split a record into parts, we so far lack means to express the complementary operation: merging two records into one. This is the essence of synchronisation in asynchronous data flow computing S-NET-style. A comprehensive study of the subject can be found in [8], but we skip it here as it is not required to understand the examples later in the paper.

III. DISTRIBUTED S-NET

As a high-level coordination language, S-NET in general is not bound to any memory model. The language concepts, however, fit in rather well with the idea of message passing. S-NET boxes and networks are indeed asynchronous components that communicate with each other by sending messages via communication channels. In principle, the language could be used to define distributed memory systems as it is by mapping components directly to nodes of the system. However, direct mapping of components may not be sensible as we must take the cost of data transfers between nodes into account. Execution times of components may vary significantly from simple filters performing lightweight operations to boxes consisting of heavy computations. Another obstacle is the dynamic nature of S-NET networks that evolve over time due to serial and parallel replication.

What we need instead of a one-to-one mapping of boxes to compute nodes is a veritable distribution layer within an S-NET network where coarse-grained network *islands* are mapped to different compute nodes while within each such node networks execute using the existing shared memory multithreaded runtime system [5]. Each of these islands consists of a number of not necessarily contiguous networks of components that interact via shared-memory internally. Only S-NET streams that connect components on different nodes are implemented by means of message passing. From the programmer's perspective, however, the implementation of individual streams on the language level by either shared memory buffers or distributed memory message passing is entirely transparent.

In principle, it would be desirable if the decomposition of networks into islands would be transparent as well, thus resulting in a fully implicit parallelisation architecture, that balances itself autonomously as the network evolves over time. With our shared memory runtime system, we have done exactly this. However, given the substantial cost of inter-node data communication in relation to intra-node communication between S-NET components the right selection

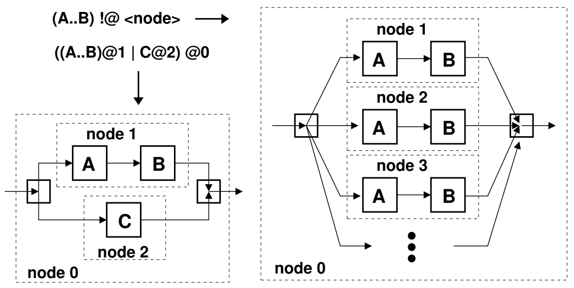


Figure 2. Example applications of static placement (left) and indexed dynamic placement (right), where we assume the tag <node> to feature values between 1 and some upper limit

of islands is crucial to the overall runtime performance of a network. Therefore, we postponed the idea of an autonomously dynamically self-balancing distributed memory runtime system for now and instead carefully extend the language in order to give the programmer control over placement of boxes and networks.

We extend S-NET by two placement combinators that allow the programmer to map networks to processing nodes either statically or dynamically based on the value of a tag contained in the data. Let A denote an S-NET network or box. Static placement (written A@42) maps the given network or box statically to one node, here node 42. A location assigned to a network recursively applies to all of those subnetworks and boxes within the network whose location is not explicitly specified by another placement combinator. If no location is specified at the outermost scope of S-NET network definition hierarchy, a default location, zero, is used instead. Fig. 2 shows an example of static placement.

The second placement combinator is an extension of the indexed parallel replication combinator. Instead of building multiple local instances of the argument network, it distributes those instances over several nodes. Let A denote an S-NET network or box, then A!@<*tag*> creates instances of A on each node referred to by <*tag*> in a demand driven way. Effectively, this combinator behaves very much like regular indexed parallel replication, the only difference being that each instance of A is located on a different node. Fig. 2 shows an example of dynamic placement.

Placement combinators split a network into sections that are located on the same node; each node may contain any number of network sections. Sections located in the same node are executed in the same shared memory, which means that data produced in one section can be consumed in another section on the same node without any data transfers between address spaces.

The concept of a node in S-NET is a very general one, and its concrete meaning is implementation-dependent. We use ordinal numbers as the least common denominator to identify nodes. These nodes are purely logical; any concrete mapping between logical nodes identified by ordinal numbers and physical devices is implementation dependent. The motivation for this is that defining the actual physical nodes in the language level would bind the program to the exact system defined at compile time. Using logical nodes allows the decisions about the physical distribution to be postponed until runtime. With MPI as our current middleware of choice the number directly reflects an MPI node. In more grid-like environments it may be more desirable to have a URL instead. We consider this mapping of numbers to actual nodes to be beyond the scope of S-NET.

IV. DISTRIBUTED S-NET RUNTIME SYSTEM

As mentioned earlier we chose MPI as middleware for its wide-spread availability and because it satisfies our basic needs for asynchronous point-to-point communication. Each Distributed S-NET node is mapped to an MPI task; the node identifier directly corresponds to the MPI task rank. Accordingly, we leave the exact mapping of logical nodes to physical resources to the MPI implementation. The Distributed S-NET runtime system is built as a separate layer on top of our existing shared memory runtime system [5]. None of the existing components of the shared memory runtime system is actually aware of the distributed memory layer. To ensure scalability, nodes cooperate as peers: there is no central control or service in the system that could become a performance bottleneck.

On the language level placement can be applied to any network or box. Hence, the placement of S-NET components onto nodes of a distributed system essentially follows the hierarchical or inductive specification of S-NET streaming networks. In general, any placement divides the network into three sections: one that remains on the original node, one that is mapped to the given node and one that is again mapped onto the original node. Of course, placement can recursively be applied to any subsection of these three network sections. As a consequence, each node hosts multiple contiguous network sections that are independent of each other. Due to parallel composition, such a network section is not necessarily a SISO component itself, but may have multiple input or multiple output streams. Each network section internally makes use of the shared memory runtime system of S-NET [5].

S-NET runtime components never send records to other nodes. Components are not even aware of nodes and the distributed runtime system. Node boundaries are hidden within specific implementations of streams. To manage streams that cross node boundaries each node runs two active components: an *input manager* and an *output manager*, as illustrated in Fig. 3. The output buffer of one network section and the input buffer of the subsequent network section can be considered as instances of the same buffer on different nodes. Output and input managers transparently move records between these buffers and take of the necessary data marshalling and unmarshalling.

Both managers are implemented by multiple threads, one per connection. This is not just a convenience with respect to exploitation of concurrency, but in fact a necessity to ensure the absence of deadlocks. In the shared memory runtime system streams are implemented as bounded FIFO buffers. Their boundedness is an important property that enforces a back propagation of resource pressure. This makes an S-NET streaming network make progress in the absence of centralised control. We carry over this idea to our distributed runtime system. Once the capacity of a distributed buffer (i.e. one that interconnects two network sections mapped to different nodes) the corresponding threads of the output manager of the first network section and the input manager of the second network section block and, hence, resource pressure is propagated back over node boundaries transparently. With multithreaded input and output managers only individual network connections block while the managers themselves remain responsive to communication requests on other inter-node connections.

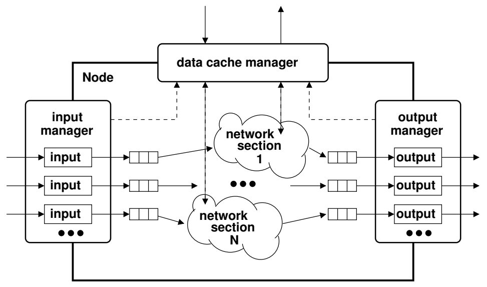


Figure 3. Internal organisation of one node

In addition to a dynamic number of *input threads* (i.e. one per inbound stream) the input manager has one *control thread*. The control thread snoops for requests to create new network sections on the node. Remember that due to dynamic serial replication and dynamic indexed parallel replication S-NET streaming networks actually evolve at runtime. When dynamic network replication expands over multiple nodes to to placement combinators in the replicated networks, a corresponding control message arrives on the node and is taken care of by the input manager's control thread, which in turn initiates network instantiation on the local node using the corresponding features of the shared memory runtime system. In addition, the control thread creates a new input thread to implement the inbound communication network connection of the new S-NET streaming network section as well as a new *output thread* of the output manager that takes care of the routing of outgoing records of the new network section and routes them to the node that hosts the subsequent network section.

In a naive approach data attached to record fields would be serialised alongside the records themselves whenever a record moves from one node to another. This obviously inflicts high overhead due to marshalling and unmarshalling of potentially large data structures and puts high demand on network performance of a distributed system. It is also generally undecidable even at runtime whether data really needs to be sent to the node hosting a follow-up network section. In particular due to flow inheritance, records typically piggy-back data that has been produced by earlier network stages and/or will be needed by later network stages that network sections in between are not even aware of and, hence, are not needed on nodes that execute such intermediate network sections.

To avoid unnecessary data transfers we completely separate data management from stream management. Data associated with record fields is never transferred between the nodes alongside the records themselves. Instead, only a representation of the data, consisting of the field label, the *unique data identifier (UDI)* of the data and the current location of the data are sent. The data itself is always fetched on demand only when a box has unpacked the required fields from an incoming record and is about to process the corresponding data. A third active component, the *data manager*, controls the movement of data in a Distributed S-NET. As illustrated in Fig. 3, the data management is one additional communication instance of each node in Distributed S-NET.

A node's data manager organises all remote fetch, copy and delete operations transparently to the rest of the runtime system. Having such a unique component on each node ensures that, for example, repeated fetch operations to identical data are avoided. References to all data elements are stored into a hash table named *data storage* that allows us to track data elements currently residing on a node. UDIs are used as hash table keys for searching specific data elements. In a way, our data management system resembles a software cache only memory architecture (COMA), where the data elements are freely replicated and migrated to the nodes' local memories.

Fetching data on demand from a remote node obviously delays the execution of a box that is otherwise ready to process. Here, it becomes apparent why we reuse our fully-fledged shared memory runtime system on each node although individual nodes may only expose a limited amount of hardware concurrency: multithreading effectively hides the long latencies of data fetch operations. For more details on the design and implementation of S-NET in general and of Distributed S-NET in particular see [4].

V. CASE STUDY: PIPELINED SIGNAL PROCESSING

Our first case study is from the area of signal processing: *Moving Target Indication* (MTI) using *Space Time Adaptive Processing* (STAP) [6]. The data-flow graph of the MTI application is sketched out in Fig. 4. Representative for signal processing applications we can identify a pipeline of filters applied in some order to a sequence of data samples. Parts of the pipeline are bypassed by certain data or alternative (sub-)pipelines are taken under certain circumstances, usually depending on properties of the processed data. In Fig. 4 boxes with folded bottom right corners denote S-NETnetworks, small boxes with text denote processing functions, and boxes containing an arrow and a capital X with a number denote structure transformers where data storage is re-arranged without affecting actual values.

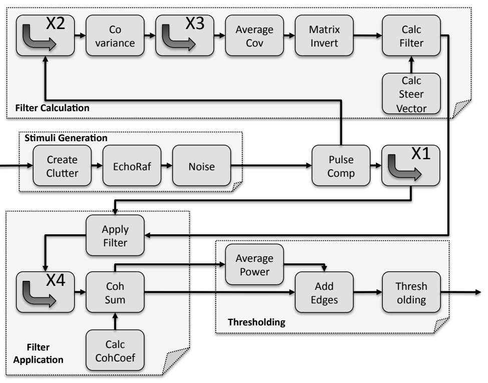


Figure 4. Data processing graph of the MTI application

The S-NET implementation of MTI can directly be derived from the data flow graph. Each signal processing function becomes an S-NET box that we derive from existing components. Then we define compound networks using combinators according to the required connections. This hierarchical approach allows us to implement and test networks independently, as each network is a fully functional application itself and can be deployed individually.

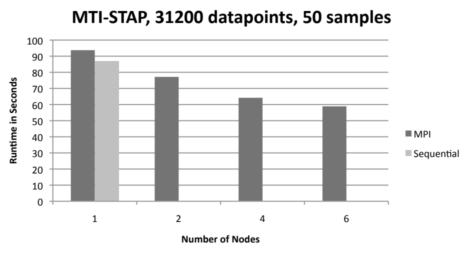


Figure 5. Runtime measurements comparing the original sequential C code with a Distributed S-NET coordination of C-implemented boxes taken from the original code on a cluster of 2, 4 and 6 nodes

The measurements we present in Fig. 5 compare the original, sequential C implementation with our S-NET implementation on a 6-node cluster of dual-processor nodes equipped with Intel PIII 1.4GHz CPUs connected via 100Mbit ethernet. Both programs were given 50 input samples and for each set the total runtime was recorded. We have made use of the static placement combinator to divide the top-level computational pipeline into 2, 4 and 6 sections. This requires only a minimal change of the original S-NET code. In fact, the high-level message passing approach of Distributed S-NET proves indispensable when it comes to finding the right places where to split the complex computational pipeline based on empirical evidence rather than guessing. Any more low-level toolset would have made the cost of implementing multiple distributions and comparing their relative virtues prohibitive. A comprehensive presentation of the MTI application including further runtime figures can be found in [7].

VI. Case Study: Client-Server

As a representative of a client-server application we use a very simple dictionary-based password cracker. It takes a dictionary and a number of Md5-encoded passwords as its input and produces the corresponding decoded password for each entry that can be cracked with the given dictionary. The cracking is done by encrypting words of the dictionary one by one and comparing the resulting hash value with the encoded password. We use the standard `glibc` function `crypt` to perform the relevant computations.

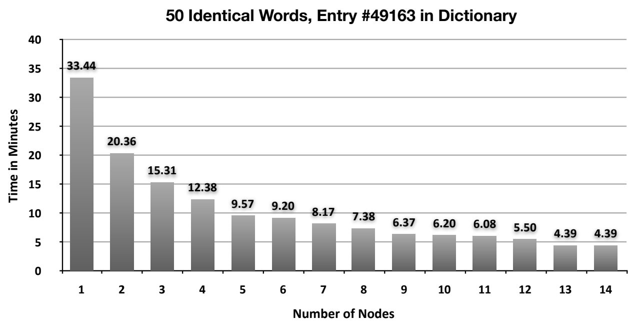
```
net decrypt ({dict, crypt_word, <nodes>, <branches>}
             -> {crypt_word, clear_word} | {crypt_word})
{
  net counter ({} -> {<cnt>});

  net balancer
      connect [{<cnt>, <nodes>, <branches>}
               -> {<node = node % nodes>,
                   <branch = (num / nodes) % branches>}];

  box cracker ((crypt_word, dict)
                -> (crypt_word, clear_word)
                 | (crypt_word));
}
connect counter .. balancer
          ..  (cracker ! <branch>) !@ <node>;
```

Figure 6. Distributed client-server style password cracker

Fig. 6 shows the complete Distributed S-NET implementation. We define a network `decrypt` that expects records with two fields (dictionary and the word to be decrypted) and two tags (the number of nodes and the number of cores per node) and that yields records that either consist of the encrypted password and its clear text value or just the encrypted password if cracking failed.

The `decrypt` network essentially is a three-step pipeline: a subnetwork `counter` adds a unique, increasing number to each record that passes through, a subnetwork `balancer` takes this number to compute both the node and the branch within that node based on the total numbers of nodes and branches chosen externally, and finally a box `cracker` that performs the main computational task. The most interesting aspect of the network `decrypt` is the wrapping of the `cracker` box within an indexed parallel replication combinator to implement branching per node and again the wrapping of that subnetwork within an indexed dynamic placement combinator that maps computations across nodes. This is all code that is needed to effectively use a two-level compute architecture made up from a network of multi-core machines.

For evaluation we use the same cluster as in the previous section. Fig. fig:runtime-cracker shows runtimes obtained on the same cluster as previously used employing different numbers of nodes in two different settings: First, we repeatedly try to crack the same word, which creates a very balanced workload. In the second experiment, we use a sequence of randomly chosen words from the dictionary, which results in a rather unbalanced workload.



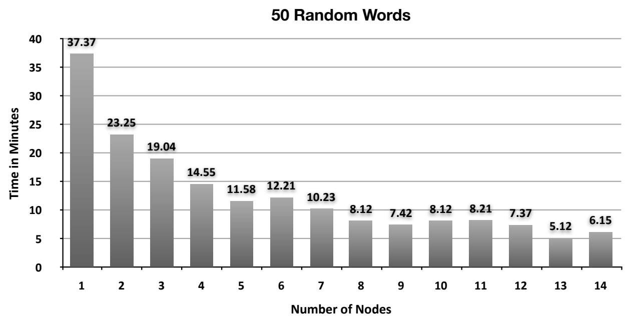


Figure 7. Speedups for client-server application: password cracking

VII. Case Study: Domain Decomposition

As our third case study we investigate the suitability of Distributed S-NET for domain decomposition models of computation. At the heart of our solution is pipeline consisting of a *splitter* that decomposes input data into a sequence of chunks, a *solver* that applies some computation to an individual chunk and a *merger* that rebuilds the output data by assembling a sequence of chunks. Fig. 8 illustrates this idea as a Distributed S-NET program. We use indexed dynamic placement to distribute the solver over a number of nodes assuming that the solver dominates the computation.

The splitter is implemented as a single box that expects records with three element: the data to be decomposed (`data`), some auxiliary data (`aux`) that is used in a read-only fashion, but may affect the solver, and the number of nodes (`<nodes`). The box `split` splits the data into chunks and outputs a stream of records in response to each single input

```
net domain_decomp
{
  box split( (data, aux, <nodes>)
                   -> (chunk, aux, <node>, <cnt>)
                    | (chunk, aux, <node>));
  box solve( (chunk, aux)  -> (chunk));

  net merge( {chunk, <cnt>} -> {data},
             {chunk}        -> {data});
} connect split .. solve!@<node> .. merge
```

Figure 8. Design of a simple domain decomposition model

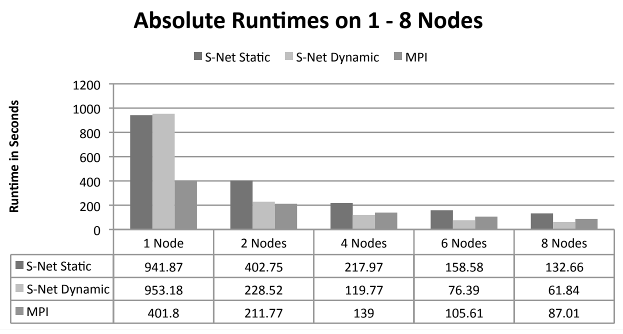
```
net domain_decomp_dynamic
{
  box split( (data, aux, <nodes>)
                   -> (chunk, aux, <node>, <cnt>)
                    | (chunk, aux, <node>)
                    | (chunk, aux));
  net compute
  {
    box solve( (chunk, aux)  -> (chunk));
    net split
      connect [{chunk, <node>} -> {chunk};{<node>}];
  } connect (((solve..split) !@ <node> | [])
        .. ( [|{chunk},{<node>}|] | []))*{chunk};

  net merge( {chunk, <cnt>} -> {data},
             {chunk}        -> {data});
} connect split .. compute .. merge
```

Figure 9. Domain decomposition model with dynamic load balancing

record. Each output record contains the unmodified auxiliary data, an individual chunk of data, representing an independent subcomputation and a tag `<node>` that determines the node that is going to process this chunk. In addition, the very first record output by the `split` box is additionally tagged with the number of chunks produced (`<cnt>`). This tag will later be used in the merge process. The box `solve` represents the essential computation that processes a chunk of data (potentially) making use of the auxiliary data provided. We use the indexed dynamic placement combinator to map these computations to different compute nodes. The merge process cannot be implemented by a single box as it combines a sequence of chunks into a single piece of data. This step requires a network and the use of synchrocells. For space limitations we leave out the definition here and refer the interested reader to [8] for more information.

The domain decomposition pattern laid out in Fig. 8 maps chunks to nodes in static way, and we generally assume that the number of chunks equals the number of nodes to be used in the computation, although that is not required technically. This scheme is likely to yield suboptimal performance if the computational complexity of processing individual chunks diverges or the compute nodes employed are heterogeneous. Fortunately, the high-level message passing approach of Distributed S-NET makes it fairly easy to extend the static mapping towards a dynamic, availability-driven mapping of M chunks to N compute nodes with $M > N$. Our solution is shown in Fig. 9.

Both the static and the dynamic domain decomposition



	1 Node	2 Nodes	4 Nodes	6 Nodes	8 Nodes
■ S-Net Static	941.87	402.75	217.97	158.58	132.66
■ S-Net Dynamic	953.18	228.52	119.77	76.39	61.84
■ MPI	401.8	211.77	139	105.61	87.01

Figure 10. Runtimes of a Distributed S-NET-parallelised ray tracer with C-implemented boxes compared with a hand-coded MPI-implementation using identical sequential building blocks

patterns have been used to implement a distributed ray tracer [9]. Experiments comparing both approaches sketched out before with a C implementation that was manually parallelised using MPI led to the results shown in Fig. 10. Note that all three implementations share the same computationally relevant code to allow for a reasonably fair comparison. We use the same workstation cluster as in the previous experiments. We always run two MPI tasks per node to compensate the fact that Distributed S-NET programs automatically exploit per-node parallelism. The runtimes on one single node clearly show the overhead added by the S-NET when compared to the MPI implementation. However, from only two nodes onwards the overheads quickly amortise. Furthermore, the dynamic load distribution scheme clearly pays off with superior parallel performance.

VIII. RELATED WORK

The coordination aspect of S-NET is related to a large body of work in data-driven coordination [10]. An early approach that treats coordination and computation as strictly orthogonal concerns is Linda [11]. Implementations of the Linda model can be found for a variety of programming languages [12], [13].

For the stream processing aspect of S-NET the language SISAL [14] needs to be acknowledged; it pioneered high-performance functional array processing with stream communication. Also functionally based is the language Hume [15], which is primarily aimed at embedded and real-time systems. Hume has a coordination layer that wires boxes into a rather synchronous streaming network. Last not least, Eden [16] is an extension of the lazy functional language Haskell. Here streams are lazy lists produced by processes defined in Haskell using a process abstraction and explicitly instantiated, which are coordinated using a functional-style coordination language.

Another recent advancement in coordination technology is Reo [17]. The focus of the language Reo is on streams, but it concerns itself primarily with issues of channel and component mobility, and it does not exploit static connectivity and type-theoretical tools for network analysis.

S-NET shares the underlying concept of stream processing with a number of *synchronous* streaming languages, such as Esterel [18] or StreamIt [19]. Asynchronous computing and the separation of concerns between computing and coordination set S-NET apart from those approaches.

Apart from programming languages we acknowledge integrated problem solving environments for scientific computing, e.g. SciRun [20]. These are graphical environments that allow the construction of simple data flow style applications based on standard component models for distributed computing. They show a surprising similarity with graphical representations of S-NET, the difference being that we use graphical notation merely for the sake of illustration for a component network itself described as data flow program.

IX. CONCLUSION

We extended the S-NET data flow coordination language by two new network combinators in order to support distributed memory architectures with inter-node communication based on message passing. Static and indexed-dynamic placement combinators allow programmers to partition any S-NET network over multiple compute nodes. The S-NET runtime system automatically deals with two levels of concurrency: semi-explicit coarse-grained concurrency on the level of a cluster or grid nodes with distributed memory and message passing communication and fully implicit fine-grained concurrency within compute nodes based on shared memory communication [5].

Writing distributed application with Distributed S-NET requires minimal programming effort, but nonetheless achieves satisfactory performance results. This opens an avenue for exploiting clusters and grids to non-experts. For the more experienced programmer, the ease of exploring the (normally) vast design space of distribution strategies allows to come up with distribution strategies that were simply not feasible with respect to implementation effort when using low-level techniques, such as MPI or PVM.

ACKNOWLEDGMENT

We thank Alex Shafarenko and Sven-Bodo Scholz for many fruitful discussions. This work was funded by the European Union through projects ÆTHER and ADVANCE.

REFERENCES

[1] C. Grelck, S. Scholz, and A. Shafarenko, "Asynchronous Stream Processing with S-Net", *Int. J. Parallel Programming*, vol. 38, no. 1, pp. 38–67, 2010.

[2] C. Grelck and S.-B. Scholz, "SAC: A functional array language for efficient multithreaded execution", *Int. J. Parallel Programming*, vol. 34, no. 4, pp. 383–427, 2006.

[3] M. Broy and G. Stefanescu, "The algebra of stream processing functions", *Theoretical Computer Science*, vol. 258, no. 1-2, pp. 99–129, 2001.

[4] C. Grelck, A. Shafarenko (eds), "S-Net Language Report 2.0", University of Hertfordshire, School of Computer Science, Hatfield, United Kingdom, Tech. Rep. 499, 2010.

[5] C. Grelck and F. Penczek, "Implementation Architecture and Multithreaded Runtime System of S-Net", in *Implementation and Application of Functional Languages*, LNCS 5836, 2011.

[6] F. Le Chevalier and S. Maria, "Stap processing without noise-only reference: requirements and solutions", *Radar, 2006. CIE'06*, pp. 1–4, 2006.

[7] F. Penczek, S. Herhut, C. Grelck, *et. al.*, "Parallel signal processing with S-Net", *Procedia Computer Science*, vol. 1, no. 1, pp. 2079 – 2088, 2010.

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

[9] F. Penczek et. al., "Message Driven Programming with S-Net: Methodology and Performance", *International Conference on Parallel Processing Workshops*, pp. 405–412, 2010.

[10] G. A. Papadopoulos and F. Arbab., "Coordination models and languages", in *Advances in Computers*. Academic Press, 1998, vol. 46, pp. 329–400.

[11] D. Gelernter, "Generative communication in Linda", *ACM Trans. Program. Lang. Syst.*, vol. 7, no. 1, pp. 80–112, 1985.

[12] E. H. Siegel and E. C. Cooper, "Implementing distributed Linda in Standard ML", School of Computer Science, Carnegie Mellon University, Tech. Rep., 1991.

[13] G. C. Wells, A. G. Chalmers, and P. G. Clayton, "Linda implementations in Java for concurrent systems", *Concurr. Comput.: Pract. Exper.*, vol. 16, no. 10, 2004.

[14] J. T. Feo, D. C. Cann, and R. R. Oldehoeft, "A report on the sisal language project", *J. Parallel Distrib. Comput.*, vol. 10, no. 4, pp. 349–366, 1990.

[15] G. Michaelson and K. Hammond, "Hume: a functionally-inspired language for safety-critical systems", in *Trends in Functional Programming*, vol. 2, 2000.

[16] R. Loogen, Y. Ortega-Mallén, and R. Peña-Marí, "Parallel functional programming in Eden", *J. Func. Prog.*, vol. 15, no. 3, pp. 431–475, 2005.

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

[18] G. Berry and G. Gonthier., "The Esterel synchronous programming language", *Sci. Comp. Prog.*, vol. 19, pp. 87–152, 1992.

[19] M.I. Gordon *et al*, "A stream compiler for communication-exposed architectures", in *ASPLOS'02, San José, USA*, 2002.

[20] K. Zhang, K. Damevski, S. Parker, "SCIRun2: A CCA framework for high performance computing", *IPDPS'04*, IEEE Computer Society, 2004.