Distributed S-Net: Cluster and Grid Computing without the Hassle

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. Components themselves are implemented using a conventional language suitable for the application domain.

We present the design and implementation of Distributed S-Net, a conservative extension of S-Net geared towards distributed memory architectures ranging from many-core chip architectures with hierarchical memory organizations to more traditional clusters of workstations, supercomputers and grids.

Three case studies illustrate how to use Distributed S-Net to implement different models of parallel execution. 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

Cluster and grid computing has traditionally been the realm of highly skilled and educated experts working in specialized application domains, where getting the highest possible performance out of a given computing platform is imperative. Consequently, fairly low-level tools prevail while programming productivity is a lesser (if any) concern.

The historic end of clock frequency scaling and the move towards multi-core/many-core chip architectures have brought shared memory parallel computing from a similar niche to the main stream. The same is likely to happen for cluster and grid computing. Whereas today’s commodity processors with small cores counts still come with shared memory, it is somewhat clear that a substantial increase in core numbers, as envisioned by the manufacturers, can only achieve scalable performance with a hierarchy of (distributed) memories. Intel’s new 48-core single chip cloud computer (SCC) illustrates this likely scenario already today.

With cluster and grid computing platforms on-chip and beyond becoming the norm rather than the exception, new programming tools and methodologies are urgently needed that effectively broaden the scope of potential applications and redefine the trade-off between hardware utilization efficiency and programming productivity in favour of the latter.

S-Net [1], [2] 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, stateless components communicating via uni-directional streams. Fig. 1 shows an intuitive example of an S-Net streaming network. The choice of a component implementation language solely depends on the application domain of the components. In principle, any programming language can be used, but for the time being we provide interface implementations for the functional array language SAC [3] and for a subset of ANSI C.

Distributed S-Net is a careful and conservative extension of S-Net that allows programmers to map sections of streaming networks onto nodes of a distributed memory on-chip or off-chip cluster or grid environment. While the programmer takes responsibility for the spatial mapping of computations all other aspects of parallel program organization (e.g. synchronization and communication) are taken
care of by Distributed S-NET. This allows programmers to harness the power of such computing infrastructures without particular effort and expertise.

The remainder of the paper is organized as follows. In Section II we provide background information to the design and rationale of S-NET. In Section III we introduce Distributed S-NET while Section IV sketches out its implementation principles. Three case studies demonstrate how Distributed S-NET can be used to implement typical models of parallel program organization: signal processing pipelines (Section V), client-server software architectures (Section VI) and domain decomposition (Section VII). Eventually, we discuss some related work in Section VIII and conclude in Section IX.

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 synchronization 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 organized 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,

$$\text{box } \text{foo} \ (\{a, b\} \rightarrow \{c\} \mid \{c, d, e\})$$

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 emphasizes 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 t1 is a subtype of t2 if t2 ⊆ t1. This subtyping relationship extends nicely 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 ∈ x is a subtype of some variant w ∈ 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 expects. 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. serialization, 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 pipeline.

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 best matches the type of the record.

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 \( \lambda \). Fig. 1 illustrates serial replication as a feedback loop. However, serial replication precisely means the repeated instantiation of the operand network \( \lambda \) and, thus, defines a streaming network that evolves over time depending on the data processed.

Indexed parallel replication (denoted \( \lambda!\langle \text{tag} \rangle \)) replicates instances of \( \lambda \) 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 summarize we can express the S-NET sketched out in Fig. 1 by the following expression:

\[
(A|B) \ldots (C!\langle t \rangle)^{\ast\{p\}} \ldots D
\]

assuming previous definitions of \( A, B, C \) and \( D \). The choice of network combinators was inspired by Broy’s and Stefanescu’s network algebra [4].

While any box can split a record into parts, we so far lack means to express the complementary operation: merging two records into one. For this purpose S-NET features *synchrocells* that capture the essence of synchronization in asynchronous data flow computing. A comprehensive study of the subject can be found in [5], but we skip it here as it is not required to understand subsequent examples.

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, S-NET could be used to program distributed memory systems right away by identifying boxes with compute nodes and sending records over streams with message passing. However, such a direct approach has two major drawbacks. Firstly, we need to take into account the cost of data transfers between nodes into account, which demands co-location of communication-intensive boxes on the same node and immediately leads to a non-trivial mapping problem. 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. Secondly, compute nodes today expose a significant degree of internal concurrency, e.g. multi-socket, multi-core, hyper-threaded systems. Identifying boxes with such nodes makes internal concurrency unavailable to S-NET, which is for obvious reasons not what we want.

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 multi-threaded runtime system [6]. 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 via 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.

It would be desirable if the decomposition of networks into islands would be transparent as well, but then networks would require to autonomously balance themselves among compute nodes as networks evolve over time through replication. 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 of islands is crucial to the overall runtime performance of a network. Therefore, we postpone the idea of an autonomously dynamically self-balancing distributed memory runtime system for now and instead carefully extend the language in order to give programmers control over placement of boxes and networks.

![Figure 2](image-url)  
**Figure 2.** Example applications of static placement (left) and indexed dynamic placement (right), where we assume the tag \(<\text{node}>\) to feature values between 1 and some upper limit.

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 \( A@0 \)) maps the given network or box statically to one node, here node 42. A location assigned to a network recursively applies to all subnetworks and boxes within the network unless their location is defined by further placement combinators. If no location is specified at the outermost scope of S-NET network definition hierarchy, a default location (node 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 \( \lambda \) denote an S-NET network or box, then \( \lambda!@<\text{tag} > \) creates instances of \( \lambda \) on each node referred to by \( <\text{tag} > \) in a demand driven way. Effectively, this combinator behaves very much like regular indexed parallel replication, the only difference being that each instance of \( \lambda \) is located on a different node. Fig. 2 also 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 on 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 Distributed 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 [6]. 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 typically 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 [6].

S-NET runtime components never send records to other nodes. They 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 output manager and an input 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 instances of the same buffer on different nodes. Output and input managers transparently move records between these buffers and take care of the necessary data marshalling, data unmarshalling and flow control.

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 centralized 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) is exhausted, the corresponding threads of the output manager of the first network section and of the input manager of the second network section block. Hence, resource pressure is transparently back-propagated over node boundaries. With multi-threaded input and output managers only individual network connections block while
the managers themselves remain responsive to communication requests on other inter-node connections.

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 due to placement combinators in the replicated network, a corresponding control message arrives at the node. It is taken care of by the input manager’s control thread, which initiates the necessary 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 serialized 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. 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. Network sections in between are typically not aware of flow-inherited data. Thus, some data associated with a record may not be needed across an entire network section mapped to some compute node.

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, only references suitable to retrieve the data when demand for the data is actually detected. Such references consist of the field label, the unique data identifier (UDI) of the data and the current location of the data. Data is only fetched on demand when a box has unpacked the required fields from an incoming record and is about to process the corresponding data. As illustrated in Fig. 3, a third active component, the data cache manager, controls storage and movement of data in Distributed S-NET.

A node’s data cache manager organizes 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. UIDs 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 proceed. 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: multi-threading effectively hides the long latencies of data fetch operations.

More details on the design and implementation of (Distributed) S-NET can be found in [2].

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) [7]. The purpose of MTI is to detect slow moving objects on the ground using a radar antenna based on an aircraft. This example application as well as the C-implementations of boxes originate from a collaboration with Thales Research, France. All details about this application can be found in [8].

The top-level 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 conditions, usually depending on properties of the processed data. In Fig. 4 boxes with folded bottom right corners denote S-NET networks, small boxes with text denote processing functions and boxes

![Figure 4. Top-level data processing graph of MTI application](image-url)
containing an arrow and a capital X with a number denote
data transformers where data storage is re-arranged without
affecting actual values.

The S-NET implementation of MTI directly mimics the
data flow graph. Each signal processing function becomes an
S-NET box whose implementation is derived from existing
code with marginal adaptation. Then we define compound
networks using combinators according to the required con-
nections. This hierarchical approach allows us to implement
and test networks independently, as each network is a fully
functional application itself and can be deployed individu-
ally.

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 1, 2, 4 and 6 nodes. The single node
“cluster” runtime shows the overhead inflicted by S-NET, i.e. the combined
overhead of the multi-threaded on-node runtime system instances and the
MPI-based inter-node runtime system fully set-up but effectively not used.

The measurements we present in Fig. 5 compare the origi-
nal, sequential C implementation with our S-NET implemen-
tation on a 6-node cluster of dual-processor nodes equipped
with Intel PIII 1.4GHz CPUs connected via 100Mbit eth-
ternet. 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 tool-set would have
made the cost of implementing multiple distributions and
comparing their relative virtues prohibitive.

We did not manage to divide the signal processing
pipeline into more than 6 sections without degrading perfor-
mance. This is an application-characteristic value. In order to
efficiently utilize larger clusters we need to combine pipeline
parallelism with additional replication of pipeline boxes.
With the high-level approach of S-NET this could be done
fairly quickly. However, we first study such approaches in
isolation in the subsequent case studies. A comprehensive
presentation of the MTI application including further run-
time figures can be found in [8].

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] | [crypt_word]}
| counter {} |
| counter {} |
| counter {} |
| counter {} |

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 implementa-
tion; a graphical illustration is given in Fig. 7. We define
a network decrypt that expects records with two fields (the
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.

```
node 0

-----

counter
balancer

-----

cracker

-----

cracker

-----

cracker

-----

cracker

-----

node n

-----

-----

-----

-----

-----


```

Figure 7. Illustration of the network presented in Fig. 6

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 utilize a two-level compute architecture made up from a network of multi-core machines.

Fig. 8 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.

VII. CASE STUDY: DOMAIN DECOMPOSITION

Our third case study looks at domain decomposition models of computation. At the heart of our solution is a pipeline of parallel computation modules that apply some computation to individual chunks and a merger that rebuilds the output data by assembling a sequence of chunks. Fig. 9 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 elements: 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 record. Each output record contains the unmodified auxiliary data, an individual chunk of data, and a tag <node> that determines the node that is foreseen 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. Indexed dynamic placement maps these computations to different compute nodes.

The merge network 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 again refer the interested reader to [5].

The domain decomposition pattern laid out in Fig. 9 maps chunks to nodes in a static way. We assume that the number of chunks equals the number of nodes, although technically this is not required. The scheme is likely to yield suboptimal performance if the computational complexity of processing individual chunks diverges or the compute nodes 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; the complete solution is shown in Fig. 10.
We employ both domain decomposition patterns (static and dynamic) to implement a distributed ray tracer [9]. Experiments compare both S-NET solutions with a C implementation that is manually parallelized using MPI; Fig. 11 shows results. Note that all three implementations share the same computationally relevant codes, and we use the same workstation cluster as in the previous experiments. For a reasonably fair comparison we only run one solver per node and use the remaining cores for auxiliary tasks. Likewise, the S-NET code does not make use of hierarchical parallel replication. In S-NET this would be easy to achieve as demonstrated in the previous section.

![Absolute Runtime on 1 - 8 Nodes](image)

Runtime on a single node show considerable overhead added by S-NET when compared to the MPI implementation. Despite some analysis we have not come to a convincing explanation for this behaviour and presume some potentially unrelated artefact to be the culprit. From two nodes onwards Distributed S-NET achieves about the same performance levels as the hand-coded MPI implementation, which implements a similar workload distribution scheme. The Distributed S-NET implementation that makes use of the dynamic load distribution pattern clearly outperforms the hand-coded MPI solution. In principle, an equivalent dynamic load balancing scheme could be implemented in MPI as well, and it would be likely to again achieve similar performance as the S-NET solution. The point, however, is that using Distributed S-NET dynamic load balancing is achieved with marginal programming effort and expertise.

### 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 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 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. However, we use graphical notation merely for the sake of illustration, whereas networks are defined using textual notation.

### IX. CONCLUSION AND FUTURE WORK

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 with negligible expertise and marginal programming effort. The S-NET runtime system automatically deals with two levels of concurrency: fully implicit fine-grained concurrency within compute nodes based on shared memory communication and semi-explicit coarse-grained concurrency in between nodes of a cluster or grid.

Writing distributed application with Distributed S-NET requires minimal programming effort, but nonetheless achieves satisfactory performance results. This opens an avenue to non-experts for exploiting cluster and grid computing infrastructures that normally require expert skills. For the more experienced programmer, the ease of exploring the (normally) vast design space of distribution strategies allows to come up with solutions that were simply not feasible with respect to implementation effort when using low-level message passing.
Various directions of future work seem attractive. The existing Distributed S-NET runtime system could be extended by an automatic work distribution and component mapping capacity that automates the placement of networks and boxes on compute nodes. As we deliberately orthogonalise mapping decision and mapping implementation, the entire technology described so far can be carried over to fully automatic solution straightforwardly.

The absence of state in box computations and the functional nature of S-NET could also be used to monitor box execution for the purpose of resilience: upon failure detection any box computation can easily be restarted on a different compute node without affecting the integrity of the Distributed S-NET as a whole. This approach opens an avenue to resilient computing, again requiring only marginal expertise and effort by programmers.

ACKNOWLEDGMENTS

We take the opportunity to thank Alex Shafarenko and Sven-Bodo Scholz for many fruitful discussions on the design of Distributed S-NET and the anonymous reviewers for their constructive comments. The work on (Distributed) S-NET has been supported by the European Commission through the projects ÆTHER and ADVANCE.

REFERENCES


