Distributed S-Net

High-Level Message Passing without the Hassle

Clemens Grelck
University of Amsterdam
Institute of Informatics
Science Park 107
Amsterdam, The Netherlands
c.grelck@uva.nl

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

Frank Penczek
University of Hertfordshire
Science and Technology
Research Institute
Hatfield, United Kingdom
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.

Categories and Subject Descriptors D.1.3 [*PROGRAMMING TECHNIQUES*]: Concurrent ProgrammingDistributed programming; D.3.3 [*PROGRAMMING LANGUAGES*]: Language Constructs and FeaturesConcurrent programming structures; D.3.4 [*PROGRAMMING LANGUAGES*]: ProcessorsRun-time environments

General Terms design, languages, performance

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

1. Introduction

The historic end of clock frequency scaling and today's hardware trend towards multi-core/many-core chip architectures [1, 2] 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 [3] 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.

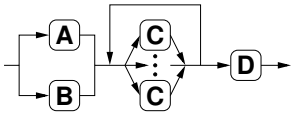


Figure 1. Illustration of an S-NET streaming network of asynchronous components

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

AMP'10, June 6, 2010, Toronto, Canada.

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 [4] 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.

The remainder of the paper is organised as follows. In Section 2 we provide background information to the design and rationale of S-NET. In Section 3 we introduce Distributed S-NET while Section 4 sketches out its implementation principles. Three case studies demonstrate how Distributed S-NET can be used to implement typical models of parallel program organisation: signal processing pipelines (Section 5), client-server software architectures (Section 6) and domain decomposition (Section 7). Eventually, we discuss some related work in Section 8 and conclude in Section 9.

2. 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 terminology, 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, respectively. As a consequence, instances of `A` and `B` operate asynchronously in a pipelined fashion. In the intuitive example of Fig. 1 serial composition can be identified between the left, the middle and the right subnetworks.

Parallel composition (denoted `(A|B)`) constructs a network where all incoming records are either sent to `A` or to `B` and the resulting record streams are merged to form the overall output stream of the compound 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. The example network in Fig. 1 features parallel composition in combining `A` and `B`. If both branches in the streaming network match equally well, one is selected non-deterministically. More precisely, the routing of such a record is under-specified and, hence, implementation-dependent. While in principle an implementation could send all such records to the, say, left branch, a more useful implementation employs some statistical distribution. However, we deliberately do not specify properties of such a statistical distribution in the language definition for now.

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, it is not. Indeed, serial replication 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. In Fig. 1 we can identify parallel replication of network `C`.

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 [5].

While any box can split a record into parts, we so far lack means to express the complementary operation: merging two records into one. Therefore, S-NET features dedicated *synchrocells*. A synchrocell has the syntactic form `[|type,type|]`. Similar to serial replication the types act as patterns for incoming records. A record that matches one of the patterns is kept in the synchrocell. As soon as a record arrives that matches the other pattern, the two records are merged into one, which is forwarded to the output stream. Incoming records that only match previously matched patterns are immediately forwarded to the output stream. Hence, a synchrocell becomes an identity after successful synchronisation and may be removed by a runtime system. The extremely simplified behaviour of synchrocells captures the essential notion of synchronisation in the context of streaming networks. More complex synchronisation behaviours, e.g. continuous synchronisation of matching pairs in the input stream, can easily be achieved using synchrocells and network combinators. See [6] for more details on this and on the S-NET language in general.

3. 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 [7]. 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 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.

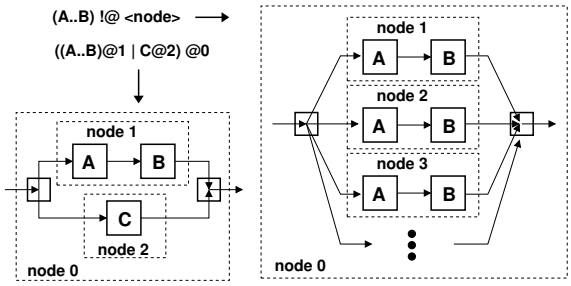


Figure 2. Example applications of static placement (left) and indexed dynamic placement (right), where we assume the tag `<node>` to feature values between 1 and some upper limit

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.

4. Distributed S-Net Runtime System

As mentioned earlier we chose MPI as middleware for its widespread 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 implemen-

tation. The Distributed S-NET runtime system is built as a separate layer on top of our existing shared memory runtime system [7]. 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 [7].

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.

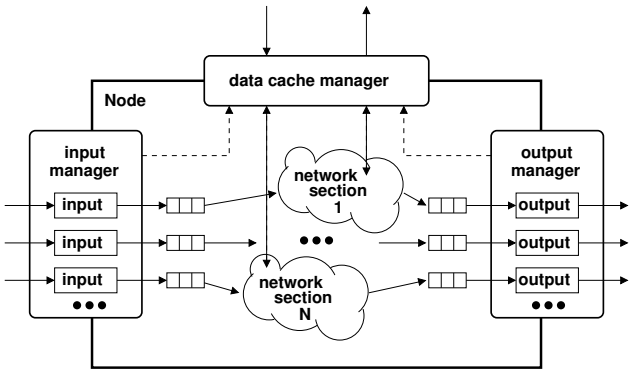


Figure 3. Internal organisation of one node

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 [8].

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 [6].

5. Case Study: Pipelined Signal Processing

Our first case study is an application from the area of signal processing. Signal processing applications are mainly characterised by a potentially deep computational pipeline of filters that are applied

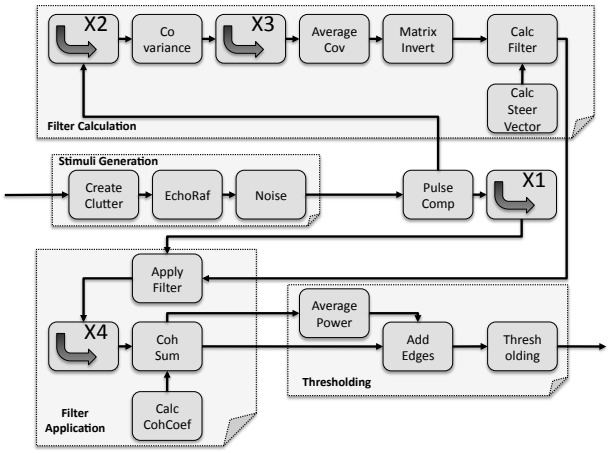


Figure 4. The data processing graph of the MTI application is subdivided into several modules. Modules are indicated by boxes with folded bottom right corners. Small boxes with text denote processing functions, boxes containing an arrow and a capital X with a number denote structure transformers. These boxes are used to re-arrange data in a matrix without effecting the actual values.

in some order to sequence of data samples. Of course, in non-trivial appliations parts of the pipeline can be bypassed by certain data or alternative (sub-)pipelines may be taken under certain circumstances, usually depending on properties of the processed data.

Our signal processing application implements *Moving Target Indication* (MTI). The purpose of MTI is to detect slow moving objects on the ground using a radar antenna based on an aircraft. The aircraft illuminates the ground with a beam orthogonal to its movement. The sequences of periodic pulses (bursts) may vary in timing and amount of bursts. The reception time of an echoed pulse depends on the number of the pulse and the distance of the reflecting surface on the ground. Measuring the latter is achieved by sampling the received signal at a given frequency, resulting in the distance being sampled into range gates (rg). The detected signal is received by the radar processing chain as a 3D array with dimensions $Nrg, Nrec, Nant$. One of the main challenges in MTI applications is to distinguish actual objects on the ground from reflections of the earth's surface (clutter). We use a technique known as 'Space Time Adaptive Processing' (STAP) [9], which computes a set of filters from signals received by the antenna array at different time steps. This example application as well as the C-implementations of boxes originate from a collaboration with Thales Research, France; more details can be found in [10].

The processing chain of the MTI application is subdivided into independent modules, as shown in Fig. 4. For simulation purposes, we also implement a 'Stimuli Generation' module, which simulates the signal received by the radar antenna array. This is achieved by computing a 2D array representing the Radar Cross Section (RCS) of the ground surface situated on range gate *rg* and angle θ. The clutter model of 'CreateClutter' is computed from random, positive values with a given average and adding peak reflectivity values of a given probability. The returned signal from a burst of pulses of the ground surface to which targets with a given RCS and radial velocity have been added, is computed by 'EchoRaf'. The final processing step in this module is the addition of white noise to the signal.

The presented processing chain contains some naive, well-known radar processing techniques for legacy reasons. Nevertheless, the characteristics, i.e. the main challenges from an implementers point of view, are representative for the important industrial domain of embedded signal-processing applications on parallel hardware, as: a) The processing chain uses multiple operators with different requirements on precision and/or dynamic ranges. b) The static processing graph represents a dynamic processing chain, as algorithm parameters, such as array sizes, loop boundaries, etc., change (multi-mode radar [11]). c) The computational load is high enough to require parallel computing hardware. d) Performance is one of the key requirements, both in terms of computational throughput and latency, which may be due to operational requirements or architecture constraints such as memory limitations.

The starting point of the design process of the MTI application in S-NET is the data-flow graph of the original implementation shown in Fig. 4. We use the structure of this graph to derive the structure of our application: Each signal processing function, i.e. the small boxes in Fig. 4, becomes an S-NET box that we build from the existing components. The modules translate to individual networks which connect the boxes using combinators according to the connections within the module. This hierarchical approach allows us to implement and test networks, i.e. the modules of the application, independently, as each network is a fully functional application itself when deployed individually.

In the sequel we illustrate the S-NET design process of successively turning application modules into networks. We start with the module 'Stimuli Generation', which according to Fig. 4 contains three processing functions. The same holds for the network we implement. The boxes are arranged in a serial combination (*CreateClutter .. EchoRaf .. Noise*). This step turns sequential function composition into a computing pipeline of three asynchronous tasks, see Fig. 4.

```
net Thresholding
{
  box ApplyFilter( (3d_signal, 4d_filter)
                   -> (4d_filtered));
  box X5( (4d_filtered) -> (4d_filtered));
  box CalcCohCoeff( () -> (coh_2d));
}
connect [{4d_filter, 3d_signal} ->
                   {4d_filter, 3d_signal}; {}]
        .. ((ApplyFilter .. X5) | CalcCohCoeff)
        .. [|{4d_filtered},{coh_2d}|];
```

Figure 5. Implementation of Thresholding network

The network that implements the 'Filter Calculation' module of the MTI application is a sequence of boxes as defined by the order of tasks shown in Fig. 4. Special treatment is necessary for boxes *CalcSteerVect* and *CalcFilter*, as the former lacks an input channel, whereas the latter requires two input channels. One possibility to model this, is to assign an empty input type to box *CalcSteerVect* and place it as direct predecessor of *CalcFilter*. A record that arrives at this box triggers execution of the box without any of the record's fields or tags being read. Flow-inheritance inserts all inbound-record constituents to the resulting record, ensuring that *CalcFilter* receives all required input fields in one record. This approach, however, does not overlap computations where it would be possible: *CalSteerVect* does not require any input, and can therefore begin its computation much earlier. To do this, the box is arranged in parallel to the rest of the network and computation is triggered at the earliest possible moment, i.e. when a record arrives at the first box. The output of the box is then combined to a result record at the latest possible stage, i.e. a synchrocell merges the box's result to the result of the remaining network. This created record contains all required fields for *CalcFilter*. Fig. 6(a) illustrates these techniques.

The remaining networks implementing 'Filter Application' and 'Thresholding' use the same techniques; for brevity we refrain from describing them in detail here. Instead, we show the (textual)

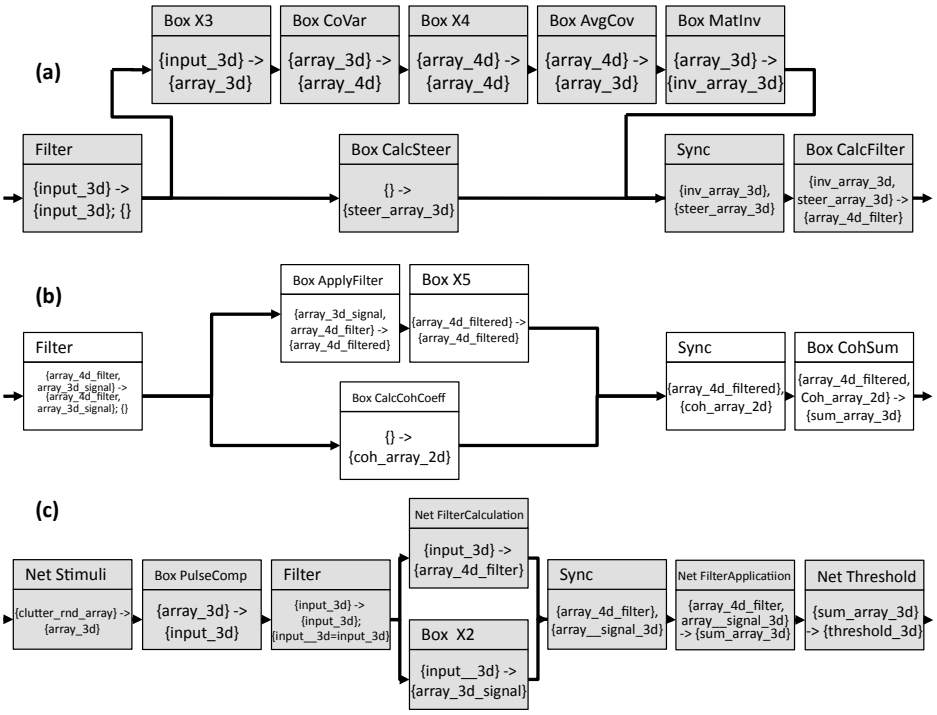


Figure 6. (a) shows network 'FilterCalculation'; (b) shows network 'Thresholding'; (c) shows the final MTI application.

S-NET implementation of network 'Thresholding' in Fig. 5 and its graphical representation in Fig. 6(b). The final step of the implementation phase is to combine all modules to form the MTI application. The complete application network is shown in Fig. 6(c).

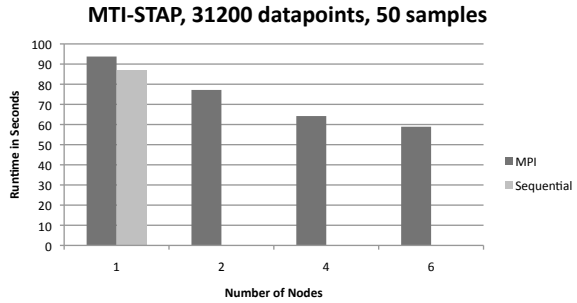


Figure 7. Runtime measurements comparing the original sequential C code with a Distributed S-NET coordination of C-implemented boxes taken from the original code on a cluster of 2, 4 and 6 nodes

The measurements we present in Fig. 7 compare the original, sequential C implementation with the S-NET implementation that we have developed. 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, respectively. 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.

These experiments were run on a 6-node cluster where each node contains two Intel PIII 1.4GHz CPUs and the nodes are connected by a standard 100Mbit ethernet network. Further runtime figures obtained by our MTI application on a 16-core shared memory compute server (not using Distributed S-NET) can be found in [10].

6. 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 8. Distributed client-server style password cracker

Fig. 8 shows the complete Distributed S-NET implementation; a graphical illustration is given in Fig. 9. We define a network `decrypt` that expects records with two fields (dictionary and the word to be decrypted) and two tags (the numbers of nodes and the numbers of branches per node) and that yields records that either consist of the encrypted password and its clear text version on successful cracking or just the encrypted password if cracking failed.

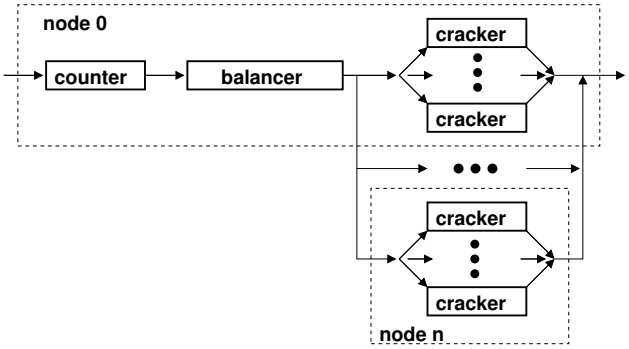


Figure 9. Illustration of the network presented in Fig. 8

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.

We leave out the definition of the `counter` network here for brevity. Its implementation is similar to the `merger` network defined in the following section. The `balancer` network is implemented using a single S-NET *filter box*. Filters are S-NET-defined boxes that do simple computations on the structure of records (e.g. removing or duplicating fields) or on the values of tags (integer arithmetic and boolean algebra). For the illustration purposes of this case study we use a simple round-robin load balancing scheme, but of course more elaborate schemes can be thought of.

For evaluation of the runtime performance we use the same cluster of dual-core nodes as in the previous section. Fig. fig:runtime-cracker shows runtimes using 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. In essence, balanced workload yields better performance, which is as expected. Still, even with unbalanced workload this simple Distributed S-NET-implemented scheme yields good results across a range of nodes.

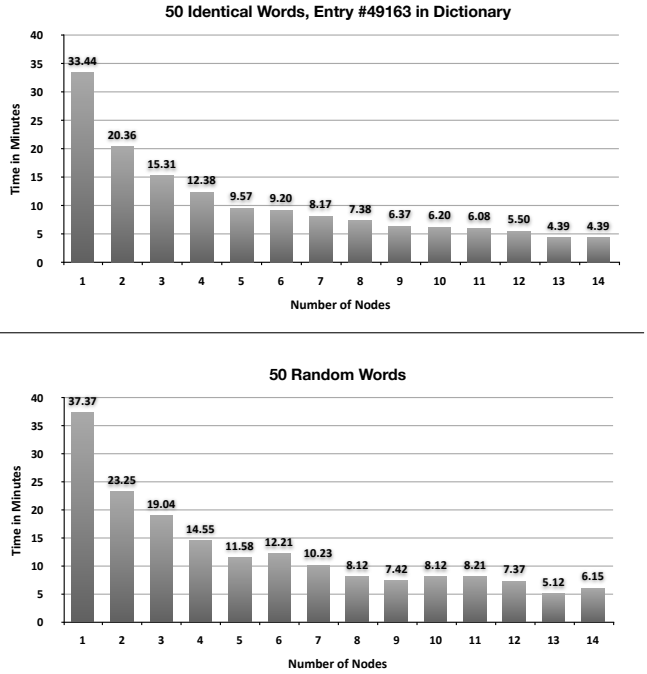


Figure 10. Runtime measurements for client-server style application: password cracking. The upper graphic shows runtimes for tasks of uniform computational complexity, whereas the lower graphic shows runtimes for tasks of random complexity

7. Case Study: Domain Decomposition

As our third and last 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. 11 illustrates this idea as a Distributed S-NET program. The main computational pipeline as described before is clearly exposed. 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 record. Each output record contains the unmodified auxiliary data, an individual chunk of data, representing an independent subcomputation and a tag `<node>` that determines

```
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 11. Design of a simple domain decomposition model

```
net merge
{
  box init ( (chunk, <cnt>) -> (accu, <cnt>));
  box step ( (chunk, accu, <cnt>) -> (accu, <cnt>)
                                   | (data));
} connect (init|[]) ..
          ([|{chunk},{accu}|] .. (step|[]))*{data}
```

Figure 12. Merger network for re-combining chunks

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.

Unlike the splitter and the solver, the merger cannot be implemented by a single box. After all, it is supposed to combine a sequence of chunks into a single piece of data. This step requires a network and the use of synchrocells. Our solution, as shown in Fig. 12 assembles two more boxes: `init` and `step`. The former turns a chunk into the final data representation ready to accommodate all further chunks (`accu`). Note that the splitter equips exactly one chunk of each original data structure with the number of chunks created (`<cnt>`). This tag ensures that only this chunk is routed through the `init` box, whereas all other chunks circumvent this box via a bypass channel.

The subsequent synchrocell recombines the accumulator with the next chunk. Note that the various chunks may arrive in the merger network in any order after their asynchronous processing by different nodes. We assume that the chunks know their location within the overall data structure. Pipelined with the synchrocell is a parallel composition. After successful synchronisation, a record has both an `accu` field and a `chunk` field. Accordingly, it is routed into the `step` box. This box implements the recombination of the chunk into the accumulator. By means of the `<cnt>` tag, the `step` box keeps track of status of the accumulator. It either outputs the enhanced accumulator in conjunction with the counter (whose value is decremented), or it outputs the final data. This distinction triggers the dynamic serial replication (the star combinator): a finalised data structure leaves the merger network, whereas an accumulator that still waits for more chunks to absorb triggers a re-instantiation of the synchrocell and the `step` box. In essence, the merger network dynamically unfolds into an `init` box followed by a number of synchrocells and `step` boxes equal to the number of chunks the data was initially split into by the splitter box. As mentioned in Section 2, a synchrocell synchronises exactly once and effectively becomes an identity box thereafter (to be garbage collected by the S-NET runtime system). This motivates the bypass of the `step` box taken by chunks that pass the synchrocell untouched.

The domain decomposition pattern laid out in Fig. 11 maps chunks to nodes in static way, and we generally assume that the

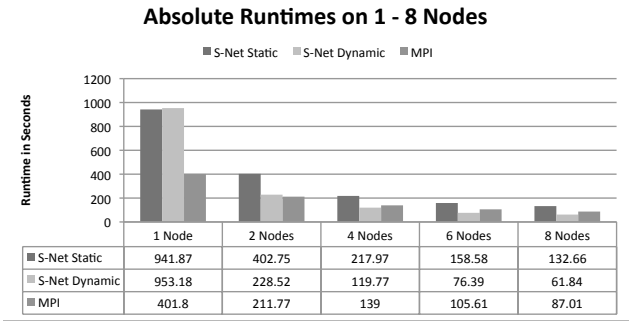
```
net domain_decomp_dynamic
{
  box split( (data, aux, <nodes>)
                -> (chunk, aux, <node>, <cnt>)
                 | (chunk, aux, <node>)
                 | (chunk, aux));
  net compute
  {
    box solve( (chunk, aux)  -> (chunk));
    net split
      connect [{chunk, <node>} -> {chunk};{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

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

Figure 13. Domain decomposition model with dynamic load balancing

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. 13.

Both the static and the dynamic domain decomposition patterns have been used to implement a distributed ray tracer [12]. Ray tracing is a fairly common technique to render a 2-dimensional image from a scene description by tracing paths of light back from the eye of an imaginary observer through pixels in an image plane to a source of light [13]. Given a scene description each pixel in the plane can be computed in isolation. This perfectly fits the domain decomposition model with the scene description as auxiliary read-only data and the chunks being subsections of the image to be computed.



	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 14. 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

Experiments comparing both approaches sketched out before with a C implementation that was manually parallelised using MPI led to the results shown in Fig. 14. 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. More details on ray tracing with Distributed S-NET can be found in [12].

8. Related Work

The coordination aspect of the proposed stream processing language is related to a large body of work in so-called data-driven coordination, see [14] for a survey of this area. An early, layered approach that, like S-NET, treats coordination and computation as strictly orthogonal concerns is Linda [15]. As S-NET, Linda is not a "complete" programming language as such, as it exclusively administers process creation and the coordination of computation which is implemented in a separate language. Implementations of the Linda model can be found for many programming languages, see [16, 17, 18] for a non-exhaustive selection. Unlike in S-NET with its stream based communication model, communication in Linda uses a shared tuple space which allows processes to interact with each other by adding, reading and removing data tuples from this shared space.

Another early source to mention is the language SISAL [19], which pioneered high-performance functional array processing with stream communication. SISAL was not intended as a coordination language, though, and no attempt at the separation of communication and computation was made in it. Still it is important to acknowledge the stream variables of SISAL as an early example of task decomposition using streams.

Also functionally based is the language Hume [20]. Hume's conceptual design is not that of a pure coordination language, but a fully-featured programming language, primarily aimed at embedded and real-time systems. Programming in Hume follows a layered approach. Values and functions are defined in a fully-functional expression language, and interaction between functions is defined in a coordination language. The finite-state machine based coordination language connects any desired amount of inbound and outbound "wires" to a function to allow for interaction between the components (i.e. the functions) of a program. Originating from Hume's primary domain and the related necessity for space- and time bound analysis [21], the expression language is an inherent part of the system and cannot be freely chosen as in S-NET. For the same reason, dynamically evolving network structures as are possible in S-NET using serial and parallel replication, are not expressible in Hume.

We also cite the work on the language Eden [22] as related to our effort, since it is based on the concept of stream communication. 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. Also, like S-NET, Eden defines a connection topology for the processing entities; it however deploys the processes completely dynamically and even allows completely dynamic channels. Eden has no provision for subtyping and does not integrate topology with types.

Another recent advancement in coordination technology is Reo [23]. 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.

Thematically closely related to the presented distributed runtime system of S-NET is the data-flow coordination language FASAN [24]. FASAN, like S-NET aims at turning regular computational functions into asynchronous stream processing components. It neither achieves nor particularly aims at a thorough separation of concerns between computation and coordination. FASAN creates a streaming network essentially through the data flow graph of a function call tree. It applies a Petri net semantics, i.e. a function becomes activated when it receives one data item on each input channel (i.e. argument) and yields exactly one data item on each output channel.

S-NET shares the underlying concept of stream processing with a number of so-called synchronous languages, such as Esterel [25] or StreamIt [26]. However, the underlying concepts — synchronous versus asynchronous stream processing — and again the separation of concerns between computing and coordination, which as such is not found in Esterel or StreamIt, make the approaches quite different in practice.

Outside the domain of high-level programming languages we acknowledge integrated problem solving environments for scientific computing, e.g. SciRun [27]. 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, whereas integrated problem solving environments take graphics first and generally lack the foundations of a programming language based solution.

9. 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. These are the static placement combinator and the dynamic indexed placement combinator. They allow programmers to partition an S-NET network over several compute nodes. As a result the runtime system deals with two levels of concurrency: coarse-grained concurrency on the level of compute nodes using distributed memory communication and fine-grained concurrency within each node using shared memory communication managed by our existing runtime system [7].

The main challenges addressed by the implementation are the dynamic construction of the S-NET network runtime representation spanning over several nodes, routing of records between the nodes and data management problems caused by the separation of the network into multiple distinct address spaces. In effect, the Distributed S-NET runtime system alone takes care of the necessary marshalling and unmarshalling of data whenever records pass node boundaries. Expensive superfluous data transfers are effectively avoided through a software COMA memory that fetches data on demand from the node that has the latest version of some data to the node node that needs to process that data next.

Three case studies demonstrate the suitability of Distributed S-NET to write real-world distributed applications: pipelined signal processing (moving target indication), client-server (password cracking) and domain decomposition (ray tracing). While message passing programming on the (system) level of MPI (or PVM) is commonly considered as very difficult, writing distributed application with Distributed S-NET requires minimal programming effort and, nevertheless, achieves more than satisfactory performance results. Furthermore, the ease of programming invites for conducting experiments to explore the (normally) vast design space of distribution strategies. For example, the choice of static network sections in the MTI application or the dynamic scheduling version of the ray tracer only become feasible because Distributed S-NET greatly facilitates such experimentation. The sheer programming effort the same experiments would take with MPI for message passing and distributed program organisation quickly prohibit or very much restrict design space exploration.

An interesting area of future research is the combination of Distributed S-NET with the ongoing research on reconfiguration and

self-adaptivity described in S-NET [28]. In conjunction, the two lines of research add further expressiveness to S-NET: distributions of networks across distributed memory environments can dynamically be changed either through external events (reconfiguration) or internal observation (self-adaptivity).

Acknowledgments

We would like to thank Alex Shafarenko and Sven-Bodo Scholz for many fruitful discussions and the anonymous reviewers for their helpful comments. This work was funded by the European Union through the Projects ÆTHER (Self-adaptive Embedded Technologies for Pervasive Computing Architectures) and ADVANCE (Asynchronous and Dynamic Virtualisation through Performance Analysis to support Concurrency Engineering).

References

[1] Sutter, H.: The free lunch is over: A fundamental turn towards concurrency in software. Dr. Dobb's Journal **30** (2005)

[2] Held, J., Bautista, J., Koehl, S.: From a few cores to many: a Tera-scale computing research overview. Technical report, Intel Corporation (2006)

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

[4] Grelck, C., Scholz, S.B.: SAC: A functional array language for efficient multithreaded execution. International Journal of Parallel Programming **34** (2006) 383–427

[5] Broy, M., Stefanescu, G.: The algebra of stream processing functions. Theoretical Computer Science **258** (2001) 99–129

[6] Grelck, C., Shafarenko, A. (eds):, Penczek, F., Grelck, C., Cai, H., Julku, J., Hölzenspies, P., Scholz, S.B., Shafarenko, A.: S-Net Language Report 2.0. Technical Report 499, University of Hertfordshire, School of Computer Science, Hatfield, England, United Kingdom (2010)

[7] Grelck, C., Penczek, F.: Implementation Architecture and Multithreaded Runtime System of S-Net. In Scholz, S., Chitil, O., eds.: Implementation and Application of Functional Languages, 20th International Symposium, IFL'08, Hatfield, United Kingdom, Revised Selected Papers. Lecture Notes in Computer Science, Springer-Verlag (2009) to appear.

[8] Protic, J., Tomasevic, M., Milutinovic, V.: Distributed Shared Memory: Concepts and Systems. John Wiley and Sons (1998)

[9] Le Chevalier, F., Maria, S.: Stap processing without noise-only reference: requirements and solutions. Radar, 2006. CIE '06. International Conference on (2006) 1–4

[10] Penczek, F., Herhut, S., Grelck, C., Scholz, S.B., Lenormand, E., Barrere, R., Shafarenko, A.: Parallel Signal Processing with S-Net. In van Albada, G., ed.: 10th International Conference on Computational Science (ICCS'10), Amsterdam, Netherlands, Elsevier Procedia Computer Science (2010) to appear.

[11] Hwang, Y., Hong, D., Kwag, Y.: Real-time o.s. based radar controller for multi-mode phased array radar system. Radar 97 (Conf. Publ. No. 449) (1997) 558–562

[12] Penczek, F., Scholz, S., Shafarenko, A., Yang, J., Chen, C., Bagherzadeh, N., Grelck, C.: Message driven programming with s-net: Methodology and performance. In: 3rd International Workshop on Programming Models and Systems Software for High-End Computing (P2S2'10), San Diego, USA. (2010) to appear.

[13] Whitted, T.: An improved illumination model for shaded display. Communications of the ACM **23** (1980) 343–349

[14] Papadopoulos, G.A., Arbab., F.: Coordination models and languages. In: Advances in Computers. Volume 46. Academic Press (1998) 329–400

[15] Gelernter, D.: Generative communication in Linda. ACM Trans. Program. Lang. Syst. **7** (1985) 80–112

[16] Siegel, E.H., Cooper, E.C.: Implementing distributed Linda in Standard ML. Technical report, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, USA (1991)

[17] Sutcliffe, G., Pinakis, J.: PROLOG-LINDA : AN EMBEDDING OF LINDA IN muPROLOG. Technical report, Department of Computer Science, The University of Western Australia, Nedlands, 6009, Western Australia (1989)

[18] Wells, G.C., Chalmers, A.G., Clayton, P.G.: Linda implementations in Java for concurrent systems: Research Articles. Concurr. Comput. : Pract. Exper. **16** (2004) 1005–1022

[19] Feo, J.T., Cann, D.C., Oldehoeft, R.R.: A report on the sisal language project. J. Parallel Distrib. Comput. **10** (1990) 349–366

[20] Michaelson, G., Hammond, K.: Hume: a functionally-inspired language for safety-critical systems. In: Draft proceedings from the 2nd Scottish Functional Programming Workshop (SFP00), University of St Andrews, Scotland, July 26th to 28th, 2000. Volume 2 of Trends in Functional Programming. (2000)

[21] Hammond, K.: Exploiting purely functional programming to obtain bounded resource behaviour: the Hume approach. In Horváth, Z., ed.: First Central European Summer School, CEFP 2005, Budapest, Hungary, July 4-15, 2005, Revised Selected Lectures. Volume 4164 of Lecture Notes in Computer Science., Springer-Verlag (2006) 100–134

[22] Loogen, R., Ortega-Mallén, Y., Peña-Marí, R.: Parallel functional programming in Eden. Journal of Functional Programming **15** (2005) 431–475

[23] Arbab, F.: Reo: a channel-based coordination model for component composition. Mathematical. Structures in Comp. Sci. **14** (2004) 329–366

[24] Ebner, R., Pfaffinger, A.: Transformation of Functional Programs into Data Flow Graphs Implemented with PVM. In: EuroPVM '96: Proceedings of the Third European PVM Conference on Parallel Virtual Machine, London, UK, Springer-Verlag (1996) 251–258

[25] Berry, G., Gonthier., G.: The Esterel synchronous programming language: Design, semantics, implementation. Science of Computer Programming **19** (1992) 87–152

[26] Michael I. Gordon *et al*: A stream compiler for communication-exposed architectures. In: Proceedings of the Tenth International Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, CA. October 2002. (2002)

[27] Zhang, K., Damevski, K., Parker, S.: SCIRun2: A CCA framework for high performance computing. In: Proceedings of The 9th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS'04), Santa Fé, NM, USA, IEEE Computer Society (2004) 72–79

[28] Penczek, F., Scholz, S.B., Grelck, C.: Towards Reconfiguration and Self-Adaptivity in S-Net. In Scholz, S.B., ed.: Implementation and Application of Functional Languages, 20th International Symposium, IFL'08, Hatfield, Hertfordshire, UK. Technical Report 474, University of Hertfordshire, UK (2008) 330–339