S-Net for Multi-Memory Multicores

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 aimed at modern multi-core/many-core architectures and systems-on-chip. 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. This two-level software architecture maintains a familiar sequential development environment for large parts of an application and offers a high-level declarative approach to component coordination.

In this paper we present a conservative language extension for the placement of components and component networks in a multi-memory environment, i.e. architectures that associate individual compute cores or groups thereof with private memories. We describe a novel distributed runtime system layer that complements our existing multithreaded runtime system for shared memory multicores. Particular emphasis is put on efficient management of data communication. Last not least, we present preliminary experimental data.

1 Introduction

Today's hardware trend towards multi-core/many-core chip architectures [24, 15] places immense pressure on software manufacturers. For the first time in history software does not automatically benefit from new generations of hardware. Today, software must become parallel in order to benefit from future processor generations! However, existing

software is predominantly sequential, and writing parallel software is notoriously difficult. So far, parallel computing has been confined to supercomputing. Now, it must go mainstream. This step requires new tools and techniques that radically facilitate parallel programming.

S-NET [12] 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 quantitive behaviour of components, it completely abstracts from (intrinsic) implementation concerns.

In fact, S-NET turns regular functions/procedures implemented in a conventional language into asynchronous, state-less components communicating via uni-directional streams. The choice of a component language solely depends on the application domain of the components itself. In principle, any conventional programming language can be used, and a single S-NET network can manage components implemented using different languages. In practice, there are, of course, limitations concerning the interoperability of languages and the technical interplay between coordination and computation layer. For the time being we provide interface implementations for the functional array language SAC [10] and for a subset of ANSI C.

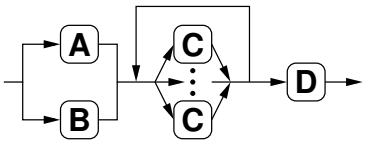

Figure 1: Illustration of an S-NET streaming network of asynchronous components

Fig. 1 shows an example of an S-NET streaming network. Note that any base component is characterised by a single input and a single output stream. This restriction is motivated, again, by the principle of separation of concerns. The concern of a box is mapping input values into output values, whereas its purpose within a streaming network is entirely opaque to the box itself. 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. Our solution achieves a near-complete separation of computing and coordination aspects. We have identified four fundamental construction principles for streaming networks:

- *serial composition* of two (potentially) different components where the output stream of one component becomes the input stream of the other;
- *parallel composition* of two (potentially) different networks where some routing oracle decides on which branch data takes;

- *serial replication* of a single network where data is streamed through the same network a dynamically determined number of times; and
- *indexed parallel replication* of a single network where an index attached to the data determines which branch (or which replica of the network) is taken.

These four construction principles allow concurrency engineers to define complex streaming networks of asynchronous components and, thus, to turn sequential code blocks into a parallel application.

While today small scale multicore processors with shared memory prevail, it is very unlikely that memory remains shared if the number of cores grows as predicted. It is time now to consider a more complex memory architecture to address likely future scenarios. In our memory model multiple cores share the same memory, but communication between one such group of cores and all other cores is by means other than shared memory. More precisely, we assume some form of network on the hardware side and message passing on the software side. This two-tier memory organisation is of interest for essentially two reasons. Firstly, for now the number of cores per chip is still very limited, only clusters of multicores provide the compute power required by challenging applications. Secondly, as soon as the number of cores on a chip exceeds a certain limit, any shared memory is likely to produce a performance bottleneck and future multicores most likely will have a memory organisation similar to our model described in order to deliver scalable performance. Contemporary manycore architectures (e.g. NVidia CUDA) already demonstrate this by a number of different memories.

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 basic concept of programming distributed-memory systems, i.e. 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 [8]. 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. In addition to the four above mentioned construction principles of networks we add two more:

- static placement of a network on some node;
- indexed placement of a network where an index attached to the data determines the node on which that data is to be routed to.

These extensions are transparent with respect to S-NET semantics, i.e. if an S-NET program is not specifically compiled for a distributed memory environment, placement has no effect. While static placement is just ignored, dynamic placement behaves like standard indexed parallel replication.

The concept of a node in S-NET is a very general one, and its concrete meaning is implementation-dependent. We use simple integer numbers to identify nodes because that choice fits the concept of tag values in S-NET, and, thus, allows programmers to compute placements both on the coordination language level (S-NET) and on the box language level. For our reference implementation we chose MPI [13] as communication middleware, mainly for its paramount availability and well-known efficiency. Hence, S-NET nodes map one-to-one to MPI process numbers. In fact, we use very few of the MPI features in order to maintain compatibility with future, potentially more lightweight middleware implementations specifically geared at multi-memory multicores, where node numbers may well denote concrete hardware cores.

The specific contributions of the paper are

- the proposal of a conservative language extension for semi-explicit placement of networks;
- description of a distributed memory runtime system implementation on top of the existing multithreaded runtime system;
- outline of a data manager service for optimised communication;
- preliminary performance figures.

The remainder of the paper is organised as follows. In Section 2 we provide a more detailed introduction to S-NET, while Section 3 introduces a running example that we come back to throughout the remainder of the paper. Section 4 describes the language extensions of Distributed S-NET in greater detail. Sections 5 and 6 illustrate the distributed runtime system and the design of the data manager, respectively. Eventually, we provide some preliminary runtime figures in Section 7, discuss 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. Messages on these 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 the resulting records on its output stream. In the simple but common case of a one-to-one mapping between input and output records the box function's result value may determine the output record. In the general case, our *box language interface* provides a box language specific abstraction named `snet_out` to dynamically produce output records during the execution of the box function. As soon as the evaluation of the box function is complete, the S-NET box is ready to receive and process the next input record.

S-NET boxes are stateless by definition, i.e., the mapping of an input record to a stream of output records is free of side-effects. We exploit this property for cheap relocation and re-instantiation of boxes; it distinguishes S-NET from most existing component technologies. In particular if boxes are implemented using imperative languages, S-NET, however, can only guarantee that box functions actually adhere to the *box language contract* as far as the box language supports such guarantees. This is in the end the same in any functional language that supports calling non-functional code.

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. As pointed out earlier, S-NET supports four network construction principles: static serial/parallel composition and dynamic serial/parallel replication. 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), they naturally express the prevailing models of parallelism (i.e. task parallelism, pipeline parallelism, data parallelism) and, last not least, we believe that these four principles are sufficient to construct most streaming networks that prove useful on a coarse-grained coordination level. 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 [3] 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 nondeterministically. More precisely, the routing of such a record is underspecified 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.

With S-NET as described so far serial replication and feedback loop are semantically equivalent, indeed. However, S-NET also features a synchronisation primitive,

named *synchrocell* that is described in more detail further below. Synchrocells join two or more records on their input stream to form a single record on their output stream. In a feedback loop, a synchrocell would generally join records that have made different numbers of iterations through the loop. Instead, our concept of serial replication ensures that synchrocells only receive records on the same level of network instantiation.

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`. While this example remains in the abstract, concrete S-NET applications can be found in [9, 11].

We already mentioned S-NET's synchronisation component called synchrocell. It takes 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 [9] for more details on this and on the S-NET language in general.

3 Running Example

Our running example is 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. Each password is associated with a cryptographic salt to make the cracking more time-consuming. We use the standard `glibc` function `crypt` to perform the relevant computations. Fig. 2 shows the complete S-NET implementation.

The code defines a network named `crypto` that consumes records containing two fields and three tags. The field `dict` contains the dictionary and the field `entries` contains a list of all the passwords and their salts. The tags `dict_size` and `num_entries` contain the number of words in the dictionary and the number of passwords, respectively. The tag `num_branches` is used to define in how many parallel branches the

```
net crypto ({dict, entries, <dict_size>,
               <num_entries>, <num_branches>}
              -> {word, <entry>} | {<false>, <entry>})
{
   box splitter ({entries, <num_entries>}
                    -> {password, salt, <entry>});

   box cracker ({password, salt, dict, <dict_size>}
                   -> {word} | {<false>});

   net load_balancer
   connect [{<entry>, <num_branches>}
              -> {<entry>,
                  <branch = entry % num_branches>}];
}
connect splitter .. load_balancer .. cracker!<branch>;
```

Figure 2: S-Net code of our running example: password cracker

processing can be made. The network produces records that either contain the decoded word and the number of the password or a tag that indicates that the password could not be cracked.

The crypto network consists of two boxes and one subnetwork. The box `splitter` takes records that hold the field containing the passwords and their salts and the tag representing the number of passwords and splits these records into smaller records, each holding fields for one password and its salt and a tag containing the ordinal number of the password. The box `cracker` does the actual password cracking. It consumes records containing the password data and the dictionary and produces decoded words or `false` tags in case the password could not be cracked. The subnetwork `load_balancer` consists of a single *filter box*. Filter boxes, or 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). The filter in Fig. 2, for example, takes records containing the ordinal number produced by the `splitter` box and assigns each record a branch number according to the ordinal number. In conjunction with the indexed parallel replication combinator around the `cracker` box, our filter realises a simple round-robin scheduler.

The records flowing in the `crypto` network are first passed in the box `splitter` which is then serially connected to the `load_balancer` network. This combination is then serially connected to the next network which is built by embedding the box `cracker` into an index split combination. The index split combinator is controlled by the tag `branch` assigned by the `load_balancer`, which means that the work is shared between `num_branches` parallel `cracker` boxes. This allows the time-consuming decoding operation to be performed in parallel to multiple passwords in case the system contains more than one processing unit.

The example also demonstrates the practical use of flow inheritance, introduced in the previous section. While the `crypto` network as a whole expects to receive records with a total of two fields and three tags, its first box (i.e. `splitter`) only expects to see one field and one tag per record. Due to flow inheritance the excess fields and tags are routed around the box itself and are transparently attached to records produced by the `splitter` box. The advantage here is that boxes like `splitter` can be defined

and implemented in a context-free manner focussing only on fields and tags that are relevant for the box itself. Still, on the coordination level the box can be integrated into a streaming network even if concrete records at some location carry additional fields and tags. As this example demonstrates, flow inheritance is a prerequisite for compositionality of streaming networks in S-NET.

4 Distributed S-Net

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.

The second placement combinator is actually 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.

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.

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.

When the placement of a network is defined, the end of the input and the beginning of the output stream of the network are always located on the given node. If the location of the network is not explicitly defined, the end of the input stream of the network is located in the node where the first component of the network is located at. Correspondingly, the beginning of the output stream of the network is located on the node in which the last component of the network is located at. The input and output streams of a network do not have to be on the same node. This feature allows an S-NET application to move data from one node to another while processing it.

```
net crypto ({dict, entries, <dict_size>, <num_entries>,
             <num_nodes>, <num_branches>}
               -> {word, <entry>} | {<false>, <entry>})
{
  box splitter ({entries, <num_entries>}
                  -> {password, salt, <entry>});

  net load_balancer ({<entry>, <num_nodes>,
                      <num_branches>}
                        -> {<entry>, <node>, <branch>})
  connect [{<entry>, <num_nodes>, <num_branches>}
              -> {<entry>, <node = entry % num_nodes>,
                  <branch = (entry / num_nodes)
                   % num_branches>}];

  net divider ({password, salt, dict,
                <dict_size>, <branch>}
                 -> {word} | {<false>})
  {
     box cracker ((password, salt, dict, <dict_size>)
                    -> (word) | (<false>));
  }
  connect cracker!<branch>;
}
connect splitter .. load_balancer .. divider !@ <node>;
```

Figure 3: Distributed S-Net specification of our running example

Fig. 3 shows a distributed version of our running example introduced in the previous section; a graphical representation of the network can be found in Fig. 4. We assume a system that consists of multiple computing nodes each of which contains a number of processing units, i.e. processors or cores. If each node had only contained a single processor, it would be straightforward to run a single `cracker` box on each node, and replacing the original index split combinator around the box `cracker` by a placement split combinator would have achieved exactly this. Consequently, we cold have achieved a distributed memory password cracker with changing only a few characters in the original S-NET code of Fig. 2.

However, assuming nodes with multiple cores, we have wrapped the box `cracker` and the index split combinator inside another subnetwork that is embedded into a placement split combinator. This solution with the help of information about the number of nodes and the updated `load_balancer` network extend the record scheduling scheme to manage multiple nodes each containing the same number of boxes. As the result of these modifications the S-NET network is spread over multiple computing nodes. The initialization tasks including the splitting of the data and the load balancer are still executed on the same node. The new `divider` subnetwork will be built into each of the nodes and the records are scheduled to each instance of the network in a round-robin fashion.

Low-level non-declarative cost intuiton would suggest that sending substantial constant data structures like the dictionary in our running example repeatedly from one node to another can hardly be efficient. However, S-NET is indeed a declarative language and the chosen specification merely says that data like the dictionary need to be present when needed. In Section 6 we will describe techniques for data management that avoid useless data transfers through runtime system support.

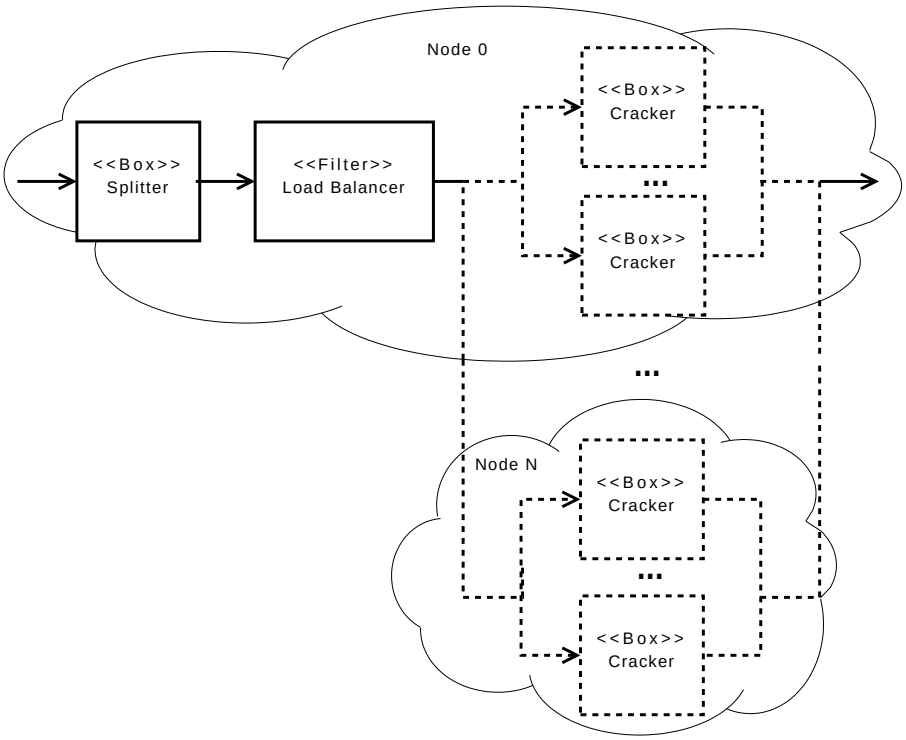

Figure 4: Illustration of the network presented in Fig. 3

5 Distributed Runtime System

The runtime support for distributed-memory systems is built as a separate layer on top of our existing shared memory runtime system. One of the main design principles is to separate these layers from each other as completely as possible. In principle, no S-NET component needs to know about the distribution as the distribution layer is entirely hidden by the realisation of streams. This design facilitates maintenance and further development of both the shared and the distributed memory versions inside the same code base.

As mentioned before we chose MPI as middleware for its wide-spread availability and because it satisfies our basic needs for asynchronous point-to-point communication and data marshalling. Each of the logical nodes is implemented as an MPI process. The logical node identifiers defined at the S-NET language level correspond directly to MPI process ranks. Accordingly, we leave the exact mapping of logical nodes to physical resources to the MPI implementation.

To ensure scalability of the S-NET runtime system implementation, the system nodes cooperate as peers: there is no central control or name servers in the system that could become a performance bottleneck. Each node is identical apart from the S-NET

components it contains.

On the language level, placement can be applied to any valid network or box. The placement combinators divide the network representation into multiple sections, each containing contiguous sequences of runtime components that are mapped into the same node. Each node may contain an unbounded number of sections like this. If a subnetwork of some network is mapped into a different node, a section is divided into multiple smaller sections. Due to parallel composition, each section may have more than one input and output stream.

The components do not send records directly to other nodes, but the boundaries between the nodes are hidden behind streams. To manage these streams each node has two active components: an input manager and an output manager. Figure 5 illustrates the architecture of a single node.

The output buffer of a section and the input buffer of the next section can be considered as instances of the same buffer on different nodes. Output and input managers transparently move records between these buffers. Both managers are implemented with multiple threads, one for each connection. The reason for this is that with blocking communication the threads can be used to propagate congestion of the streams to preceding nodes without blocking the whole node. Secondly, multi-threading is used to prevent dead-locks. S-NET implementation uses bounded-size streams to propagate congestion within the nodes. In single-threaded implementation in cases where there are two such sections in the same node that one is reachable from the other, a dead-lock may occur if the input manager blocks because of a full stream. In multi-threaded implementation a dead-lock is not possible, because only one of the inputs is blocked, not all of them. All the threads work completely independently and there is no shared state between them.

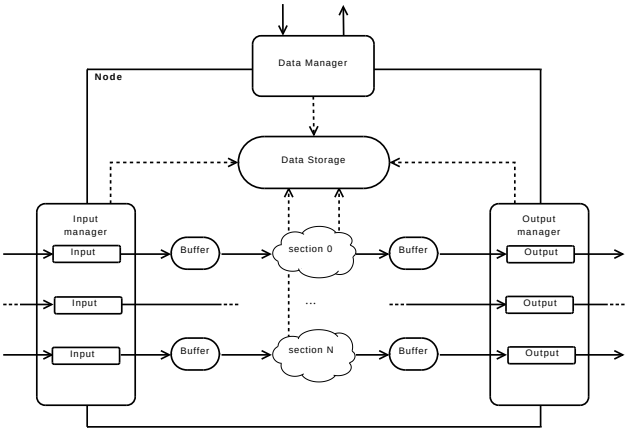

Figure 5: Internal organisation of one node

The input manager consists of one *control thread* that listens for control messages sent by the other nodes and one *input thread* per stream that arrives in to the node. The control thread listens to requests to create new network sections and update messages

that contain information about new connections. Update messages trigger creation of a new input thread. Each input thread listens to exactly one connection, deserialises incoming records and passes them to the input stream of the corresponding network section.

The output manager consists of an *output thread* per stream that leaves the node. Each thread serves as a counterpart for an input thread on some other node. Output threads simply serialise records and send them to the node containing the next section.

Data management is separated from the stream management. The box language data is not transferred between the nodes with the records. Instead only a representation of the data, consisting of the label of the field, the *unique data identifier (UDI)* of the data and the current location of the data with accuracy of a node, is sent. The real data is later fetched separately on demand. A UDI is a globally unique name for a data item, that is used to refer the data item without exact knowledge of its location in the memory.

The motivation for separation of the data and the records is that a record may flow through several nodes before a particular data element is consumed. By fetching a data element only into those nodes where it is actually needed, unnecessary data transfers can be avoided. Another motivation to separate the data and the records is that, even though both of them are moved from one node to another, the needs may be quite different. For example, records are assumed to be relatively small messages, while the size of the data elements may range from bytes to gigabytes.

Another active component, the *data manager*, is used to handle data management needs of the distributed S-NET. References to all data elements are stored into a hash table named *data storage* that allows tracking of data elements currently residing on a node. UDIs are used as hash table keys for searching specific data elements.

In general, the input manager controls all the communication between the nodes, except the communication related to remote data operations explained more in Section 6. This also gives the input manager an important role in creation of new network sections. This is discussed more in the next section.

6 Managing Data Communication

In Distributed S-NET boxes are mapped to certain nodes. Hence, data needs to be moved between the nodes. Data transfers may have a serious effect on runtime performance, depending on the underlying system, amount of data to be moved and data access patterns of the program. The programmer has the main responsibility in achieving performance and can affect the performance by choosing the right data access patterns, that is, minimising data transfers between the nodes. This section describes how the prototype implementation manages the box language data.

One of S-NET's main design principles is to separate coordination from computation. As the result, the current shared-memory implementation of S-NET is almost unaware of the box language data. The data elements are simply collected into records and managed through opaque pointers and copy and delete functions offered by the language interfaces. Only the boxes know the data representation and can operate directly on the data. In a distributed memory environment data management becomes

somewhat more complex. To be able to move data between the nodes in potentially heterogeneous environments the runtime system requires more information about the data representation.

Motivated by the performance penalty of the data transfers, the main principle of the data management is to avoid any unnecessary data movement. In the shared memory S-NET implementation each data element is directly stored into a record. In Distributed S-NET, the box language data is not transferred inside the records over node boundaries, but instead only a representation of the data consisting of the labels and locations of each data element is included. The motivation for this is that the fact that a record is passed to a node does not imply that all the data of the record is needed there. In fact, a record may flow through several nodes carrying exactly the same data, in which case most of the data transfers would be unnecessary. The tag values are always transferred with the records as they might be needed for routing purposes.

S-NET requires abstract copy and delete operations from the language interfaces, but doesn't particularly define how the operations are to be performed. The current language interface implementations for C and SAC use reference counting, which goes well hand-in-hand with S-NET's functional behaviour requirements for the boxes. In the distributed memory implementation, deep memory copies may be wasted in case the data is not used on the same node but instead transferred to another node after the copy. Reference counting is a cheaper operation in this case. On the other hand, always assuming reference counting unnecessarily limits the language interface implementation.

This problem is solved by postponing the language interface level copy until the copied data is actually needed. This is done by implementing reference counting mechanism inside the runtime system and only performing language interface level copy if there are multiple fields referring the same data when one of the fields is consumed by a box. This decision does not restrict possible future language interfaces that perform real copying instead of reference counting or inherent reference counting of any box language, as the language specific copy operation is eventually done if it is actually required.

There is also another advantage in introducing the runtime system level reference counting. In addition to avoiding unnecessary copies, also unnecessary data transfers may be avoided in some cases. Since only the boxes can modify data and because of their functional behaviour, it is safe to make assumptions about which data elements are identical. Copies of a data element can be tracked and in case a node contains a real copy of the data at the time when another fetch for the same data occurs, the fetch can be satisfied much more efficiently by using the local copy instead and simply adjusting the reference counts properly.

The opaque data pointers in records are replaced by *reference objects*. A reference object holds the label of the referred data element and accounting information like the count of fields referring to that particular data element through that reference. Data elements are identified by their UDIs. UDIs in our case consist of the identifier of the node where the data is originally created and a running counter specific to that node. UDIs never change during the life time of a reference object. As data regularly migrates, the node part of the UDI is not used for locating the data; that information is separately contained within the reference object. References in records always point

directly to the data element; no reference chains need to be collapsed upon data access. The reference objects are used to hide the reference counting and distribution of the data from the rest of the runtime system.

In S-NET it is not generally decidable where some data element is actually needed. This rules out any push communication. Only when some data is actually requested by a box, the runtime system transparently fetches the data as necessary. An exception are filters: they may copy or discard data elements, but do not require us to actually fetch the data at all. Instead we use remote copy and delete operations: rather than loading the data onto the node that executes the filter, the appropriate command is sent to the node that currently hosts the data.

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. In a way, our data management system resembles a software COMA (cache only memory architecture) where the data elements are freely replicated and migrated to the nodes' local memories [20]. Pulling data into nodes just before it is required by a box introduces delays on box processing. Here, it becomes apparent why our nodes are again multi-threaded themselves even if the number of cores per node is small or the nodes are effectively unicores. In a sequential node implementation the deferred data fetch would have an adverse effect on performance as the actual processing of data would generally be postponed until the last piece of data has arrived. With our multi-threaded node implementations we effectively hide the latencies inflicted by fetching remote data as late as possible.

A data fetch protocol consist of three MPI messages. First a request message is sent to the node where the data currently resides. This request identifies the requested data element by its UID. The data manager of that node then replies with a message containing representation of the data type of the requested data element. The data type is reconstructed in the other node and memory is allocated for the data based on the type. After this the data manager transmits the actual data. Since the S-NET runtime system is unaware of the data representations of the box language interfaces, the language interfaces play a crucial role in the data transfers. They for example manage data type and data marshalling with the help of the MPI library.

Fig. 6 illustrates data management by means of an example. the original instance of the data resides on node 1 and is locally referred to by a single record. References to the data have been carried by records to every other node in the system. None of the records referring to the data has yet been consumed on nodes 2 and 4. Both the nodes contain several records referring the data, possibly as the result of local copies. These copies have been satisfied by increasing the reference count. The local reference in these nodes points to the original reference in node 1, as there is no local copy of the real data in these nodes. In node 3 the case is somewhat different: there have been at least two records referring to the data, and one of them has been consumed. Because of this, the real data has been fetched into the node and all the local references point to the local copy instead of the original copy on node 1. In case the record on node 3 is moved, for example to node 4, the record is modified to point to the reference on that node, and the data in node 3 is deleted, as the last reference to it would be lost.

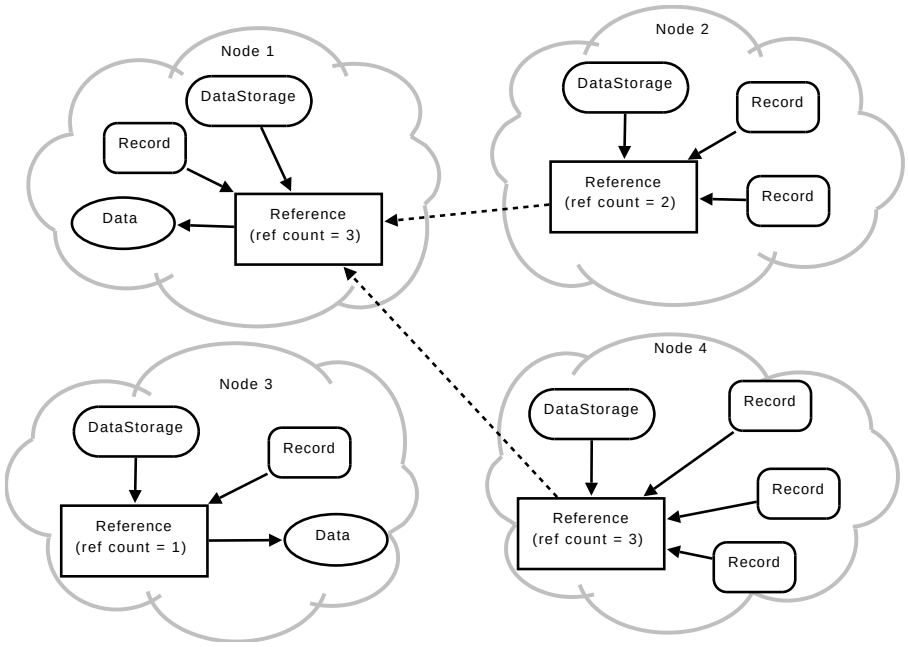

Figure 6: Data management example

7 Experimental Evaluation

We experimentally evaluate our approach using the running example of a password cracker. The boxes of the program are implemented as C language functions using S-Net's C language interface. We use Ubuntu's British-English dictionary containing about 100,000 entries.

In the absence of multi-memory multicore processors for the time being, our first test environment is a small cluster of 1.8 GHz Dual-Core AMD Opteron 1210 processors connected through Gigabit Ethernet and running Ubuntu Linux and MPICH-2 [2] with maximum threading support enabled. We investigate three problem sizes trying to crack 12, 36 and 60 passwords, respectively. We always use the same word as passwords, which generates more or less even workloads on our system.

nodes	12	36	60
1	1.006	1.002	1.002
2	0.503	0.502	0.504
3	0.337	0.337	0.337

Figure 7: Relative wall clock execution times of the running example using 1, 2 or 3 nodes and dictionaries with 12, 36 and 60 entries

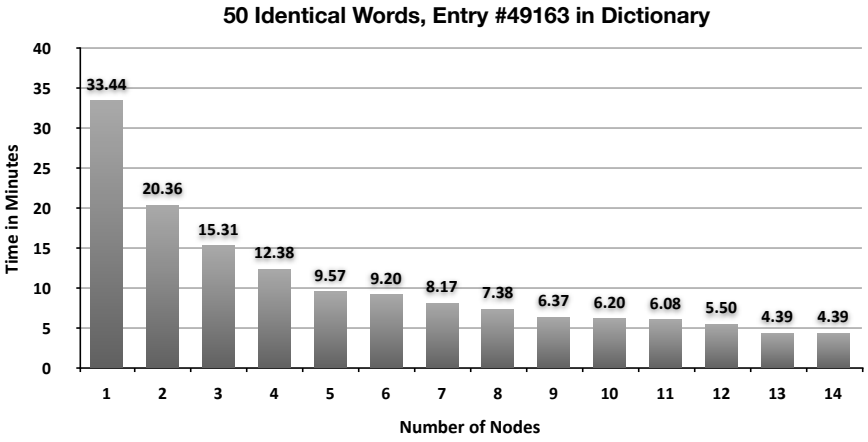

Figure 8: Measured wall clock times of the running example on 14 dual-processor compute nodes searching for the same password in the dictionary (even workload)

Fig. 7 shows relative wall clock execution times measured on our experimental setup, i.e. the execution time of a distributed version of our running example on 1, 2 or 3 nodes divided by the execution time of a non-distributed version running on a single node. We observe nearly linear speedups for all size classes. If you wonder why we do not see a 6-fold speedup given that we have a total of six cores, bear in mind that the non-distributed version used as the baseline of this experiment already makes effective use of the two cores of a single machine. Because of this the increase in computing power is only threefold.

Our second experimental setup consists of a cluster of 14 Pentium-III based dual-processor nodes running at 1.4GHz connected through 100Mbit Ethernet. We use a similar software installation as before: Ubuntu Linux and OpenMPI. Fig. 8 demonstrates that the problem scales well for this larger number of dual-processor compute nodes.

In Fig. 9 we show results from using the same experimental setup as before, but changing the password cracking problem such that we actually search for random words from the dictionary. This naturally results in an uneven workload distribution. Nevertheless, the figures demonstrate good scaling behaviour, although absolute performance degrades slightly as expected.

Indeed, our chosen application is embarrassingly parallel, but one needs to take into account that these speedups on distributed memory architectures have been achieved without any classical parallel programming involved, solely by means of S-Net coordination of conventional C-implemented components. [1]

[1] We are fully aware that both architectural setups are not ideal to demonstrate what our system is intended for, but for the time being we had no access to a larger cluster of up-to-date multicore nodes.

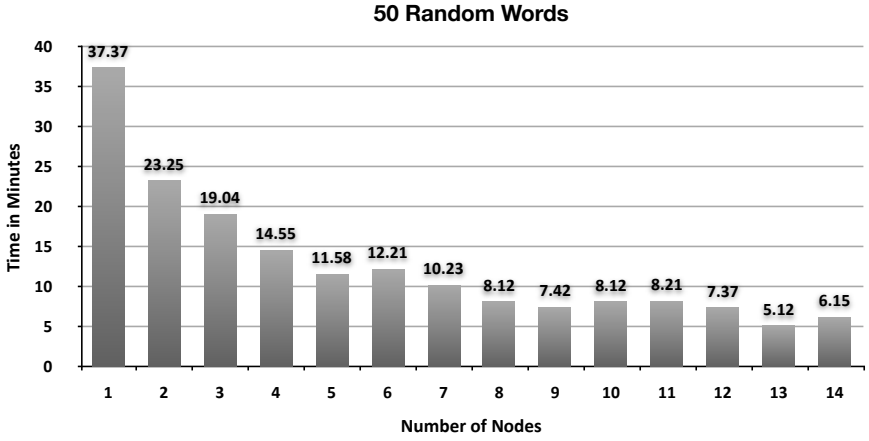

Figure 9: Measured wall clock times of the running example on 14 dual-processor compute nodes searching for a random sequence of passwords in the dictionary (uneven workload)

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 [18] for a survey of this area. An early, layered approach that, like S-NET, treats coordination and computation as strictly orthogonal concerns is Linda [7]. 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 [21, 23, 25] for a non-exhaustive selection. Unlike in S-NETwith 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.

The earliest closer related proposal, to our knowledge, is the coordination language HOPLa from the Utrecht University's Ariadne project [6]. It is again a Linda-like coordination language, which uses record subtyping (which they call "flexible records") in a manner similar to S-NET, but does not handle variants as we do, and has no concept of flow inheritance. Also, HOPLa has no static "wiring" and does not use type to establish a stream configuration.

Another early source to mention is the language SISAL [5], 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 [17]. 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 [14], 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 shall also cite the work on the language Eden [16] 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 [1]. 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 are many systems that aim to orchestrate computation in a distributed memory setting. We cite here FASAN [4], a coordination language primarily designed for recursive numerical algorithms. A FASAN program describes the data-flow graph of an application whose nodes are sequential modules written in an external computation language like C or Fortran. Distributed execution of a FASAN program is implemented using PVM [22].

Outside the domain of high-level programming languages we acknowledge integrated problem solving environments for scientific computing, e.g. SciRun [26]. 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 architectures where several cores share their memory while communication between different groups of course is 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 [8].

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. Preliminary experiments show that the approach taken allows us to achieve considerable speedups on clusters of multicore processors with a distributed memory architecture without compromising the high-level programming style of S-NET, i.e. without addressing architectural detail or low-level organisational concerns.

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 [19]. 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, Sven-Bodo Scholz, Juha Pärssinen and Juha Koivisto for many fruitful discussions on this work and the anonymous reviewers for their detailed and helpful comments. This work was partially funded by the European Union through the Integrated Project Æther (Self-adaptive Embedded Technologies for Pervasive Computing Architectures).

References

[1] Farhad Arbab. Reo: a channel-based coordination model for component composition. *Mathematical. Structures in Comp. Sci.*, 14(3):329–366, 2004.

[2] Argonne National Laboratory. *MPICH2*, 2009. http://www.mcs.anl.gov/mpi/mpich2.

[3] Haoxan Cai, Susan Eisenbach, Clemens Grelck, Frank Penczek, Sven-Bodo Scholz, and Alex Shafarenko. S-Net Type System and Operational Semantics. In *Proceedings of the Æther-Morpheus Workshop From Reconfigurable to Self-Adaptive Computing (AMWAS'08), Lugano, Switzerland*, 2008.

[4] Ralf Ebner and Alexander Pfaffinger. 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*, pages 251–258, London, UK, 1996. Springer-Verlag.

[5] John T. Feo, David C. Cann, and Rodney R. Oldehoeft. A report on the sisal language project. *J. Parallel Distrib. Comput.*, 10(4):349–366, 1990.

[6] G Florijn, T Bessamusca, and D Greefhorst. Ariadne and HOPLa: flexible coordination of collaborative processes. In P Ciancarini and C Hankin, editors, *First International Conference on Coordination Models, Languages and Applications (Coordination'96), Cesena, Italy, 15-17 April, 1996. LNCS 1061*, pages 197–214, 1996.

[7] David Gelernter. Generative communication in linda. *ACM Trans. Program. Lang. Syst.*, 7(1):80–112, 1985.

[8] C. Grelck and F. Penczek. Implementation Architecture and Multithreaded Runtime System of S-Net. In S.B. Scholz and O. Chitil, editors, *Implementation and Application of Functional Languages, 20th International Symposium, IFL'08, Hatfield, United Kingdom, Revised Selected Papers*, volume 5836 of *Lecture Notes in Computer Science*. Springer-Verlag, 2010.

[9] C. Grelck, Shafarenko, A. (eds):, F. Penczek, C. Grelck, H. Cai, J. Julku, P. Hölzenspies, Scholz, S.B., and A. Shafarenko. S-Net Language Report 1.0. Technical Report 487, University of Hertfordshire, School of Computer Science, Hatfield, England, United Kingdom, 2009.

[10] Clemens Grelck and Sven-Bodo Scholz. SAC: A functional array language for efficient multithreaded execution. *International Journal of Parallel Programming*, 34(4):383–427, 2006.

[11] Clemens Grelck, Sven-Bodo Scholz, and Alex Shafarenko. Coordinating Data Parallel SAC Programs with S-Net. In *Proceedings of the 21st IEEE International Parallel and Distributed Processing Symposium (IPDPS'07), Long Beach, California, USA*. IEEE Computer Society Press, Los Alamitos, California, USA, 2007.

[12] Clemens Grelck, Sven-Bodo Scholz, and Alex Shafarenko. A Gentle Introduction to S-Net: Typed Stream Processing and Declarative Coordination of Asynchronous Components. *Parallel Processing Letters*, 18(2):221–237, 2008.

[13] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message Passing Interface*. MIT Press, Cambridge, Massachusetts, USA, 1994.

[14] Kevin Hammond. Exploiting purely functional programming to obtain bounded resource behaviour: the Hume approach. In Zoltán Horváth, editor, *First Central European Summer School, CEFP 2005, Budapest, Hungary, July 4-15, 2005, Revised Selected Lectures*, volume 4164 of *Lecture Notes in Computer Science*, pages 100–134. Springer-Verlag, 2006.

[15] Jim Held, Jerry Bautista, and Sean Koehl. From a few cores to many: a Tera-scale computing research overview. Technical report, Intel Corporation, 2006.

[16] R. Loogen, Y. Ortega-Mallén, and R. Peña-Marí. Parallel functional programming in Eden. *Journal of Functional Programming*, 15(3):431–475, 2005.

[17] Greg Michaelson and Kevin Hammond. 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.

[18] G A Papadopoulos and F Arbab. Coordination models and languages. In *Advances in Computers*, volume 46, pages 329–400. Academic Press, 1998.

[19] Frank Penczek, Sven-Bodo Scholz, and Clemens Grelck. Towards Reconfiguration and Self-Adaptivity in S-Net. In Sven-Bodo Scholz, editor, *Implementation and Application of Functional Languages, 20th International Symposium, IFL'08, Hatfield, Hertfordshire, UK*, Technical Report 474, pages 330–339. University of Hertfordshire, UK, 2008.

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

[21] Ellen H. Siegel and Eric C. Cooper. Implementing distributed linda in standard ml. Technical report, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, USA, 1991.

[22] V. S. Sunderam. PVM: a framework for parallel distributed computing. *Concurrency: Pract. Exper.*, 2(4):315–339, 1990.

[23] Geoff Sutcliffe and James Pinakis. Prolog-linda : An embedding of linda in muprolog. Technical report, Department of Computer Science, The University of Western Australia, Nedlands, 6009, Western Australia, 1989.

[24] Herb Sutter. The free lunch is over: A fundamental turn towards concurrency in software. *Dr. Dobb's Journal*, 30(3), 2005.

[25] G. C. Wells, A. G. Chalmers, and P. G. Clayton. Linda implementations in java for concurrent systems: Research articles. *Concurr. Comput. : Pract. Exper.*, 16(10):1005–1022, 2004.

[26] K. Zhang, K. Damevski, and S.G. Parker. 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*, pages 72–79. IEEE Computer Society, 2004.