2010 39th International Conference on Parallel Processing Workshops

Message Driven Programming with S-Net: Methodology and Performance

Frank Penczek, Stephan Herhut
Sven-Bodo Scholz, Alex Shafarenko
University of Hertfordshire, UK
{f.penczek,s.a.herhut,
s.scholz,a.shafarenko}@herts.ac.uk

JungSook Yang
Chun-Yi Chen
Nader Bagherzadeh
University of California, Irvine
{jyang12,cchen29,nader}@uci.edu

Clemens Grelck
University of Amsterdam, Netherlands
c.grelck@uva.nl

***Abstract*—Development and implementation of the coordination language S-NET has been reported previously. In this paper we apply the S-NET design methodology to a computer graphics problem. We demonstrate (i) how a complete separation of concerns can be achieved between algorithm engineering and concurrency engineering and (ii) that the S-NET implementation is quite capable of achieving performance that matches what can be achieved using low-level tools such as MPI. We find this remarkable as under S-NET communication, concurrency and synchronization are completely separated from algorithmic code. We argue that our approach delivers a flexible component technology which liberates application developers from the logistics of task and data management while at the same time making it unnecessary for a distributed computing professional to acquire detailed knowledge of the application area.**

I. INTRODUCTION

The idea of representing an application as a set of components coordinated by a program written in a separate language goes as far back as the language Linda [1], but it has never achieved prominence. We argue that one reason for this may have been that with previous approaches the separation between algorithmic and coordination code has always remained incomplete. Indeed, coordinating sequential programs involves splitting, synchronizing and rejoining sequences, at which moments data is exchanged and the meaning of any part of a program potentially ceases to be self-contained. To understand the context of a unit of computation, one needs to see the coordination plan for the whole application. However, if so what is the point of writing that plan in a different language? One might as well stay with the language of choice and use, for instance, a message-passing library, such as MPI, for parallelization.

Coordination is often loosely referred to as "orchestration" thus invoking a musical analogy. If one were to follow it, one might also remark that the orchestral score is often impenetrable to a player, who has to rely on the conductor for the general musical intent. By contrast, software crucially relies on hierarchical abstraction for its maintainability; there is no conductor in the picture, instead every part has to be clear to its "player", i.e. the interpreting eye of a programmer, all by itself, without detailed knowledge of the application's "grand design". The coordination language S-NET, which we discuss in this article, wields this crucial power of encapsulation.

How is this achieved? First of all, the root cause of the mutual penetration of coordination and computation codes under any existing paradigm of coordination is the fact that the computational code *even at the unit level* contains a sequence of state transitions with at least part of the state being (potentially) exposed to coordination. As the experience accumulated in the S-NET project shows [2], this is completely avoidable. All that is required is a set of properly encapsulated units, each being a self-contained function of value parameters received only via the explicit parameter-passing mechanism, and each producing output value parameter-list messages for others. The state-transition behaviour of the functions should be hidden so that no further input into the function is possible until its termination. Such functions (called "boxes" in S-NET terminology) can be written in any language as long as it is guaranteed that they do not use any mutable static data: a new invocation of a box on the same parameter tuple must always produce the same results. Boxes can thus be relocated from processor to processor between invocations, and since they do not hold state, they can also be replicated at will.

The state-machine behaviour required for a distributed algorithm can now be achieved completely outside the boxes by a coordination language. The interface between the coordination layer and the box language involves neither concurrency aspects nor the data concept of the box language. Boxes appear in the coordination program in fully abstract form: as box names and box type signatures that list parameters without defining value sets. The extreme, total separation is not desirable since boxes may signal application-specific decisions that affect coordination. For this purpose S-NET allows them to communicate integer scalars (integers *are* the universal language of all abstract machines) to and from the coordination language, but that is all that can ever be communicated "in the clear" across the box interface.

Box execution can *always* be concurrent since boxes cannot interact with one another in the course of execution. In this sense, the execution is 100% data-driven. Data movement between boxes remains the *only* concern of the coordination language. It now encompasses all aspects of concurrency, including synchronization, throttling, threading, etc. We shall dwell a little on two peculiar aspects of data movement under coordination: topology and flow inheritance.

1530-2016/10 $26.00 © 2010 IEEE
DOI 10.1109/ICPPW.2010.61

A. Topology

is the static part of the information about message destinations. A coordination language based on boxes does not have to enforce static routing of messages, but in many cases dynamic routing would break the box abstraction. Indeed, a box function producing messages for another box would need to know where that box may be found. Unless the boxes participating in the exchange are replicas of the same box, this destroys the locality of interpretation (i.e. requires the knowledge of the aforementioned grand design for understanding a single component). On the other hand, static routing, while obviating destination information in messages, introduces unwelcome variety at the unit level of coordination: boxes may have several (different for different boxes) output routes, which have to be associated with the destination boxes by coordination languagefacilities. General network description mechanisms are quite unwieldy (see, for example, the algebra of flownomials [3]). To avoid this, S-NET limits all boxes to a single-input-single-output (SISO) configuration. The coordination layer can easily split a single stream of messages into several substreams, based on the data type and type constraints of the recipients. The converse problem, namely, merging independent data streams into one for the purposes of single input can change the concurrent behaviour of the application. The S-NET solution is to allow for nondeterministic mergers that effectively merge messages in the order of arrival thus weakening the negative effects of confluence. This way instead of offering a variety of many-to-many connectors to the coordination programmer, S-NET can limit itself to only four generic SISO-to-SISO (optionally) nondeterministic *network combinators*. As a result, network construction can be hierarchically described in S-NET by an algebraic formula, which looks somewhat similar to regular expressions in a Kleene algebra.

B. Inheritance

is the ability to expand units of application code without breaking their abstraction. The form of inheritance utilized in OOP is not very useful for coordination since OOP inheritance is object-centric. In the proposed scheme objects are stateless and transient, so the emphasis shifts back towards the processing components. Since a flat-list message assembled from opaque parameter values is the basic communication quantum, S-NET provides a subtyping solution for such messages. They are treated as opaque records, i.e. sets of label-value pairs, with the subtyping relation being the inverse set inclusion relation on label sets. For example, a component expecting a record $\{a, b\}$ can also accept $\{a, c, b\}$ (by ignoring the value of c) as a subtype of the input type. The coercion to the supertype and the formation of the argument tuple by sorting the label-value pairs in the parameter list order with a subsequent stripping of the labels happens in the coordination layer; a box function is always fed the correct value lists for box execution. One of the most potent features of S-NET is the peculiar form of binary inheritance that takes into account the pipelined nature of distributed computation. Since any record can safely be expanded with extra items, S-NET transfers the items unmatched at the box inputs to each of the output records of the box (unless an identically labeled item is included in it already, a form of override). As a result, a chain of boxes operating on a message can process a certain subset of it each, while being oblivious of (but not thus destroying) the rest of the message. This is the S-NET concept of *flow inheritance*. It is our experience that flow inheritance dramatically simplifies coordination of complex applications as it improves the abstraction and self-containment of components.

The rest of the paper is organized as follows. Section II introduces an example application, which is a ray tracing algorithm. Section III provides an outline of the S-NET language. Section IV discusses the programming of ray tracing in S-NETṪhe subsequent section contains some performance figures obtained by direct measurement. Section VI presents an account of related work and finally there are some conclusions.

II. RAYTRACING

Ray tracing is a well-known technique of rendering a 2D pixel images of a 3D scene model by tracing paths of light back from the eye of an imaginary observer through pixels in an image plane and by simulating the effects of their encounters with the objects of the scene illuminated by a source of light [4].As illustrated in Fig. 1, the primary ray is shot through each pixel in the image plane and tested for intersection against the objects in the scene. When the closest intersection is found, the pixel value is computed based on the characteristic of the object material. Rays are continually generated until either the end of the scene or the maximum ray depth level is reached.

As each ray is cast to every object, the majority of the rendering time is spent calculating intersections. In order to enable efficient ray tracing, we use the Bounding-Volume Hierarchy (BVH) algorithm [5]. It builds a hierarchical representation of 3D objects that makes the traversal of the nodes in search for intersections more efficient. More precisely, when adding an object to the BVH, it inserts the bounding volume that contains the object at the optimal place in the hierarchy using a branch-and-bound algorithm, which minimizes the cost estimation based on the surface area [6].

Algorithm 1 Ray Tracing Algorithm : it loops over the entire image, casting a single ray per pixel.

```
1: /* Input : Scene Information */
2: /* Output: 2D Rendered Image */
3: scene ← construct a Bounding Volume Hierarchy (BVH) based on the input scene
4: for each pixel in the image plane do
5:     ray ← construct the primary ray from the center of projection through pixel
6:     color ← Trace( ray, scene ) /* See Algorithm 2 */
7: end for
```

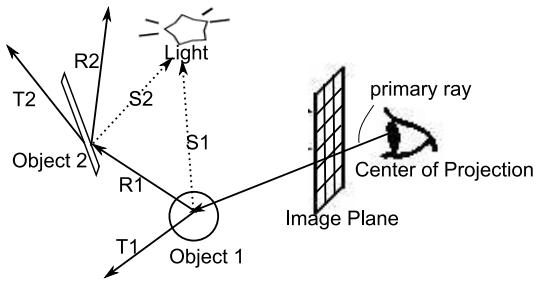


Fig. 1. Image rendering via ray tracing. A primary ray is cast and tested for intersection with objects. At the closest hit point, reflective ray R1, shadow ray S1, and transmitted ray T1 are generated, each of which is also tested for intersection with the objects in the scene.

Algorithm 2 Trace Algorithm : it follows the ray, and if it finds a closet hit, it decides the pixel color based on ray interactions with the objects. It selects the background color by default.

```
1: /* Input : ray, scene */
2: /* Output: a shade of a pixel */
3: if ray_depth < MAX_RAY_DEPTH then
4:     hitinfo ← Cast(ray, scene)
5:     if hitinfo is not null then
6:         color ← Shader(hitinfo)
7:     end if
8: end if
```

Algorithm 1 provides the pseudo code of ray tracing process that we used in this paper, and Algorithm 2 describes the procedure of tracing rays and deciding the shade of a pixel. The Cast function in Algorithm 2 traverses the BVH data structure, which contains the objects of the scene, to find the closest intersection between the ray and the objects. When it finds a hit, it calculates the shade of the pixel considering the reflective, refractive, shadow ray interactions.

The highly computation intensive nature of ray tracing and the fact that each pixel of an image can be rendered independently of all others make ray tracing amenable to aggressive parallelization. Indeed, many attempts have been made at implementing ray tracing on distributed systems and multicore architectures [7], [8]. The implementation we use in this paper distrubutes an image evenly across all cluster nodes and processes these indepentently. The root process collects all sub-results and assembles the completed scene.

III. Distributed S-Net

S-Net turns functions written in a standard programming language (C, for example) into asynchronously executed, stateless stream-processing components, termed *boxes*. Each box is connected to the rest of the network by two typed streams: one for input and one for output. Messages on these typed streams are organized as non-recursive records, i.e. sets of label-value pairs. The labels are subdivided into *fields* and *tags*. The fields are associated with values from the box language domain; they are entirely opaque to S-Net. Tags are associated with integer numbers that are accessible both on the coordination and on the box level. Tag labels are distinguished from field labels by angular brackets. Operationally, a box is triggered by receiving a record on its input stream. It applies the box function to the record. In the course of function execution the box may communicate records on its output stream. Once the execution of the box function has ended, the S-Net box is ready to receive and process the next record on the input stream.

On the S-Net level a box is characterized by a *box signature*: a mapping from an input type to a disjunction of output types. For example,

```
box foo ((a,<b>) -> (c) | (c,d,<e>));
```

declares a box that expects records with a field labeled `a` and a tag labeled `b`. The box responds with an unspecified number of records that either have just field `c` or fields `c` and `d` as well as tag `e`. The associated box function `foo` is supposed to be of arity two: the first argument is of type `void*` to qualify any opaque data; the second argument is of type `int`.

The box signature naturally induces a *type signature*. Whereas a concrete sequence of fields and tags is essential for the proper specification of the box interface, we drop the ordering when reasoning about boxes in the S-Net domain and turn tuples of labels into sets of labels. Hence, the type signature of box `foo` is `{a,<b>} -> {c} | {c,d,<e>}`. We call the left hand side of this type mapping the *input type* and the right hand side the *output type*.

To be precise, this type signature makes `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 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 $w \in y$. Again, the variant v is a subtype of w if and only if every label $\lambda \in v$ also appears in w. Subtyping on input types of boxes raises the question what happens to the excess fields and tags. As mentioned previously, S-Net supports the concept of flow inheritance whereby excess fields and tags from incoming records are not just ignored in the input record of a network entity, but are also attached to any outgoing record produced by it in response to that record. Subtyping and flow inheritance prove to be indispensable when it comes to getting boxes that were designed separately to work together in a streaming network.

It is a distinguishing feature of S-Net that it neither introduces streams as explicit objects nor that it defines network connectivity through explicit wiring. Instead, it uses algebraic formulae to describe streaming networks. The restriction of boxes to a single input and a single output stream (SISO) is essential for this. S-Net provides four network combinators: static serial and parallel composition of two networks and dynamic serial and parallel replication of a single network. These combinators preserve the SISO property: any network, regardless of its complexity, is an SISO entity in its own right.

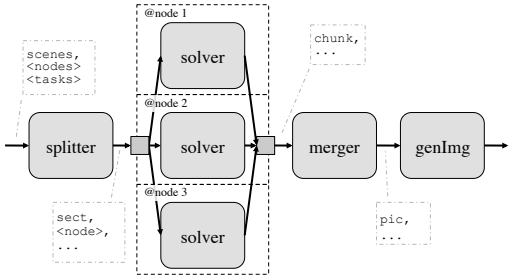
Let `A` and `B` denote two S-Net networks or boxes. Serial combination (`A..B`) constructs a new network where the

output stream of A becomes the input stream of B, and the input stream of A and the output stream of B become the input and output streams of the combined network, respectively. As a consequence, A and B operate in pipeline mode. Parallel combination (A|B) constructs a network where 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 combined network. The type system controls the flow of records. Each network is associated with a type signature inferred by the compiler. Any incoming record is directed towards the subnetwork whose input type better matches the type of the record. If both branches match equally well, one is selected non-deterministically. The parallel and serial combinators have their infinite counterparts: serial and parallel replicators for a single subnetwork. The serial replicator A*type constructs an infinite chain of replicas of A connected by serial combinators. The chain is tapped before every replica to extract records that match the type specified as the second operand. The parallel replicator A!<tag> also replicates network A infinitely, but the replicas are connected in parallel. All incoming records must carry the tag; its value determines the replica to which a record is sent.

There is one "stateful" entity in S-NET, called *synchrocell*. It provides the only means in S-NET to combine two or more existing records. Remember that the opposite direction, splitting a record into two or more records, can easily be achieved by any box. Syntactically, a synchrocell consists of comma-separated list of type patterns enclosed in [| and |] brackets, for example [| {a,b,<t>}, {c,d,<u>} |]. The synchrocell holds incoming records which match one of the patterns until all patterns have been matched. Only then are the records merged into a single one, which is released to the output stream. A match happens when the type of the record is a subtype of the type pattern. The pattern also acts as an input type specification of the synchrocell: it only accepts records that match at least one of the patterns.

As described so far, S-NET is an abstract notation for streaming networks of asynchronous components. In particular, there is no notion of computing resources in S-NET, nor does S-NET make any specific assumptions about the execution environment. Distributed S-NET [9] is a conservative extension of S-NET that introduces the concept of abstract compute nodes as an organizational layer on top of the logic network of boxes defined by standard S-NET. Again, let A denote an S-NET network or box. we introduce two additional *placement* combinators as follows. *Static placement* A@num places the box or network A onto compute node num for execution. *Indexed dynamic placement* A!@<tag> places the execution of network or box A onto the node identified by the value of tag on a per-record basis. More precisely, every incoming record is routed through a replica of A instantiated on the compute node determined by the value of tag.

We deliberately restrict ourselves to plain integer values for identifying compute nodes to retain the advantages of an abstract model as far as possible. The concrete mapping of numbers to machines is implementation-dependent. Our prototype implementation of Distributed S-Net is based on MPI where numbers correspond to MPI task identifiers.




```
net raytracing_stat
{
  box splitter( (scene, <nodes>, <tasks>)
      -> (scene, sect, <node>, <tasks>, <fst>)
         | (scene, sect, <node>, <tasks> ));
  box solver  ( (scene, sect)  -> (chunk));
  net merger  ( (chunk, <fst>) -> (pic),
                (chunk)        -> (pic));
  box genImg  ( (pic)          -> ());
} connect

splitter .. solver!@<node> .. merger .. genImg
```

Fig. 2. Overall design for a simple fork-join model.

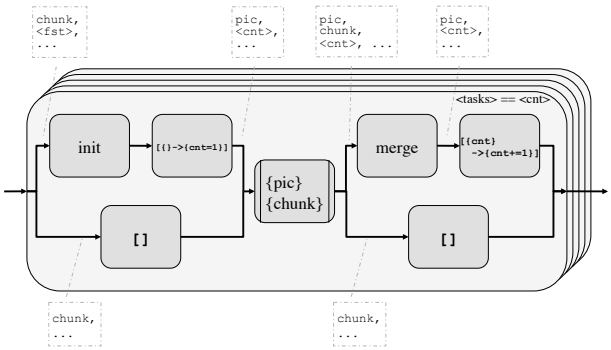
Readers are referred to [10], [11], [12] for a more thorough presentation of the general language design and to [9] for more information on the design and implementation of Distributed S-NET.

IV. RAY TRACING IN S-NET

Raytracing lends itself nicely to concurrent execution. All rays can be traced individually allowing for top-level coarse grain parallelism. To start with, we want to capture this concurrency by a simple fork-join approach.

A. A simple fork-join model in S-NET

We create three major components, a *splitter*, a *solver*, and a *merger* and we combine these in a pipeline-like fashion as shown in Figure 2. The splitter divides the overall task into sections, which are subsequently computed by instances of the solver, each of which is executed on an individual MPI node. After the solvers have dealt with the individual sections, the merger collects all the resulting image chunks into an overall result picture which is written to a file by means of a box *genImg*. Note that the entire distribution of data and re-collection of results on the master node is being triggered by the use of the @-symbol in the index splitter. It ensures that the value of the tag <node> is interpreted as the MPI-rank (processing node) that executes the corresponding solver instance. Therefore, the scheduling is controlled by the splitter component, which identifies the individual sections of work and attaches <node> tags to the records. Also note that the signatures of the individual boxes only define the parts of the data to be manipulated; the actual records may contain extra fields which are preserved by flow inheritance. The tag <fst> is an example of this.




```
net merger
{
  box init  ( (chunk, <fst>) -> (pic));
  box merge ( (chunk, pic)   -> (pic));
} connect
    ( ( init .. [ {} -> {<cnt=1>} ] )
      | []
    )
    .. ( [| {pic}, {chunk} |]
         .. ( ( merge
                  .. [ {<cnt>} -> {<cnt+=1>}]
              )
              | []
            )
       )*{<tasks> == <cnt>} ;
```

Fig. 3. Merger network for re-combining subimages into a complete picture.

Most of the components could easily be derived from the existing C-code. Using the C interface for S-NET (see [10] for details), only small wrapper functions needed to be created. The only component that could not be mapped directly to a C-implemented box was the merger. This was due to the fact that the merger needs to combine several chunks arriving asynchronously within separate records while boxes can only ever see one record at a time. Therefore, we created a sub-net named *merger* in which we specified the step-wise synchronisation of chunks in terms of two further components: an *init* box and a *merge* box. The init box creates an initial version of the resulting picture from the first chunk of work (tagged by a flag `<fst>` by the splitter component), which serves as an accumulator throughout the merging process. An n-fold application of the box merge is achieved by placing it under a serial replication combinator. Figure 3 shows the details of the merger net. The init box is followed by a filter which adds a flag `<cnt>` initialised by the value 1. This flag is used to count the number of subimages that have been incorporated into the result image already. Since only the first chunk needs to be processed by the init box, we also provide a bypass to the initialisation path for all the other records containing further chunks. This is done by means of an empty filter box which is parallel to the initialisation path.

After the initialisation, we have a star which implements the merging with the remaining chunks. In each unfolding (iteration) of the star, we first have a synchrocell, which synchronises the accumulator held in `{pic}` with yet another chunk. The resulting joint record, containing the accumulated picture and a chunk to be inserted, is presented to the merge box which outputs the combined picture. The insertion of a new chunk is reflected in an increment of the flag `<cnt>` as defined by the subsequent filter. Once the counter equals the overall number of tasks, which is kept in another, flow-inherited flag `<tasks>`, the accumulated picture is output from the merger network. The reader may wonder why we have a bypass parallel to the merge branch within the star. This is due to the fact that the star combinator does not feed any records back, but instead unrolls into copies of its operand. As a consequence, all but the first chunks not yet processed need to be bypassed to the next instance of the star.

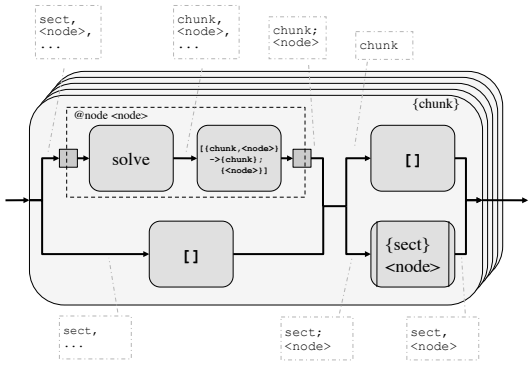
While the network presented so far serves its purpose perfectly well, imbalances in the distribution of objects within any given scene quickly lead to limited scalability on clusters with more than 2 processing nodes. To improve on this situation, a dynamic workload balancing scheme had to be put into place.

B. Fork-Join with dynamic scheduling

Here, the strict separation of application and concurrency engineering as it is enforced in S-NET pays off. Only very little modification to the basic solution described so far was required in order to achieve dynamic scheduling. The basic idea is to get the `<node>` flag to represent the availability of the processor identified by the value of that flag for computing an arbitrary section on it. We can enable the splitter box to output sections which, initially, do not contain a node flag. As a result, we now have two kinds of computing task streaming towards the solver component: those that do have a node flag attached to them, and those that do not. We can process the former straight-away by using the solve box while we need to queue the others to wait for node tokens. These node tokens can then be taken from tasks that arrive back from solvers. This modification of the S-NET solution presented so far can be achieved by simply replacing the `solver@<node>` component from Figure 2 by the network segment shown in Figure 4.

Here, we see that the solve box directly links into a filter which separates the resulting image chunk from the node token. Both these activities happen now on the individual nodes for all those sections that actually do come with a node token. All other sections just bypass the distributed solver. The reunited streams of computed chunks, node tokens, and remaining sections are then fed into either a synchrocell, which combines the remaining section with a just released node token, or a bypass, which enables the computed chunks to leave the solver segment. Those sections that have been combined with a new node flag are brought to a new instance of the distributed solver by means of a surrounding star. This unfolding of the start takes place until all sections have been transformed into chunks of the resulting picture.

Since the remaining part of the S-NET presented in Section IV-A is oblivious of the node tag, it can be utilised in the dynamic setting without modification.




```
( ( ( solve .. [ {chunk, <node>}
                 -> {chunk}; {<node>} ]
    )!@<node>
    | []
  )
  .. ( [] | [| {sect}, {<node>} |] )
) * {chunk}
```

Fig. 4. Solver segment for dynamically scheduled work load distribution.

V. Performance

We have conducted several experiments using the original C/MPI implementation and S-Net solutions with and without dynamic load balancing, and recorded the runtime of each using a scene of 3000×3000 pixels.

The experiments were run on an 8-node cluster where each node contains two Intel PIII 1.4GHz CPUs and 1024MB of RAM. The nodes are connected by a standard 100Mbit ethernet network and all nodes have access to a shared file system.

For the dynamic load balancing solution we have experimented with several scheduling algorithms and found that block scheduling and a simple variant of factoring [13] produces the best results. In the latter case, the scheduler divides the problem into several batches of sections, where in each batch the sections are of the same size. The section size decreases from batch to batch by a certain factor. For example, suppose a scene of 3000×3000 pixels is split along the y axis by dividing it into 48 section. One possible scheduling is to split the scene into two batches with the first batch containing 24 sections of size 93 and the second batch the remaining 24 section of size 32.

The results for several task sizes with varying token numbers are shown in Fig. 5(left) for factoring scheduling and in Fig. 5(right) for block scheduling. As can be seen in the diagrams, performance was generally best when 16 tokens were made available to the system. With this number of tokens each node holds two tokens on average which maps one solver instance to each CPU of the node. In the block scheduling case 32 tokens could be beneficial, but a further investigation is required to establish this. Performance is generally at its worst when the number of tasks equals the number of tokens. In this case all sections are immediately mapped to the nodes and the benefits of dynamic scheduling are lost.

To test the scalability of our approach, we compared the runtimes of all variants on one to eight nodes. Fig. 6 shows the results of these experiments. The runtimes on one single node clearly show the overhead the S-Net runtime system adds to the application when compared to the original MPI implementation (labelled MPI in the figure). However, from only two nodes onwards the overheads are amortised. For better utilisation of the computing resources we added two more static variants that spawn two instances of the solver per node (one per CPU). These experiments came at almost no additional development cost: by adding one more index split combinator to the solver of Fig. 2 (`(solver!<cpu>)!@<node>`) and marking input data with a `<cpu>` tag of values 0 and 1, the desired effect was achieved for the S-Net implementation. The MPI implementation did not require any code changes but the experiments were re-run with two processes per node by starting $2n$ MPI jobs on n nodes. The runtimes of this experiment are labelled "S-Net Static 2CPU" and "MPI 2 Proc/Node" in Fig. 6. As the S-Net runtime system automatically utilises multiple cores if available, the limited performance gain was to be expected. The MPI implementation however could benefit substantially from utilising the second CPU of each node. We also include runtimes for dynamic scheduling using number of nodes $\cdot 8$ tasks and tasks$/2$ tokens with block scheduling (labelled S-Net best dynamic) to present the compelling improvements on runtimes this technique offers. For comparison between these implementations, Fig. 6(right) shows the speed-up of the 2 CPU MPI version versus the 2 CPU static S-Net implementation and the best dynamic scheduling run.

VI. Related Work

The coordination aspect of the proposed stream processing language is related to a large body of work in data-driven coordination; see [14] for a survey of this area. An early approach that similar to S-Net treats coordination and computation as orthogonal concerns is Linda [1]. Like S-Net, Linda is not a "complete" programming language; 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; for example [15], [16], [17] to cite a few. Unlike S-Net with its stream based communication model, Linda uses a shared tuple space for communication, which allows processes to interact with each other by adding, reading and removing data tuples from the shared space.

The earliest proposal that is related more closely to our work is, to the best of our knowledge, the coordination language HOPLa from the Utrecht University's Ariadne project [18]. It is again a Linda-like coordination language, which uses record subtyping (called the "flexible records" concept) in a manner similar to S-Net, but it does not handle variants as we do, and it has no concept of flow inheritance. Also, HOPLa has no static "wiring" and does not use types to establish a stream configuration.

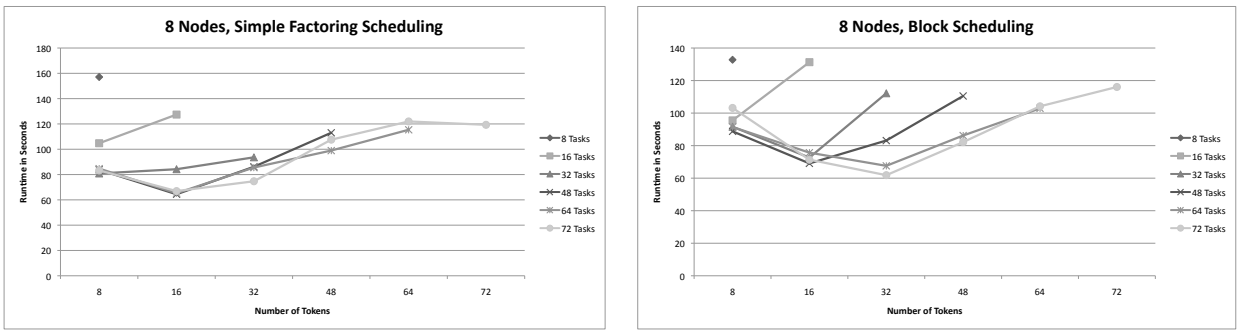
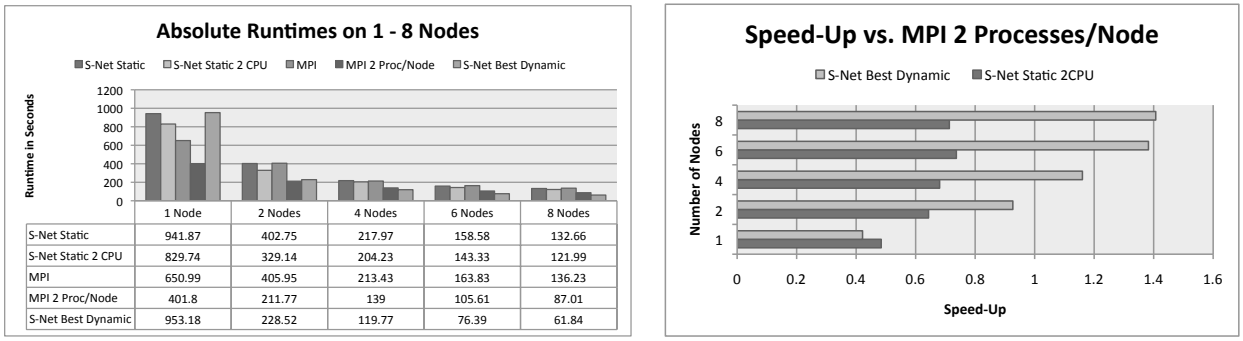


Fig. 5. Runtimes on 8 nodes using simple factoring scheduling (left) and block scheduling (right) on a 3000 by 3000 pixels scene



	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 Static 2 CPU	829.74	329.14	204.23	143.33	121.99
MPI	650.99	405.95	213.43	163.83	136.23
MPI 2 Proc/Node	401.8	211.77	139	105.61	87.01
S-Net Best Dynamic	953.18	228.52	119.77	76.39	61.84

Fig. 6. Runtimes on 1 - 8 nodes, comparing the original MPI implementation against S-NET variants (left) and speed-up of each implementation measured against the original MPI implementation with 2 processes per node (right)

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. Consequently, it makes no attempt to separate communication from computation. Still, it is important to acknowledge the stream variables of SISAL as an early example of task decomposition using streams.

Likewise 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 intended for embedded and real-time systems. Programming in Hume follows a layered approach. Values and functions are defined in a fully-functional expression language; 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 (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, which 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 [22]. It is, like S-NET, based on the concept of stream communication. In Eden streams are lazy lists produced by processes defined in Haskell using a process abstraction. They are explicitly instantiated and 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 are many systems that aim to orchestrate computation in a distributed memory setting. We cite here FASAN [24], 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.

The ongoing trend towards to parallel chip architectures and the need for parallel software outside the classical application domains have stimulated research into programming languages with explicit concurrency constructs. They are: Charm++ [25], X10 [26], Chapel [27] and Cilk++ [28], to name a few. In one way or another, they all extend a sequential base language

to express general computations by explicit concurrency management constructs. In contrast to our work on S-NET, they neither aim at a separation of concurrency and application engineering nor are they based on the concept of stream processing.

Outside the domain of high-level programming languages we acknowledge integrated problem solving environments for scientific computing, e.g. SciRun [29]. 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 illustrative purposes, whereas integrated problem solving environments take graphics first and generally lack the foundations of a programming-language based solution.

VII. CONCLUSION

A design methodology based on an extreme separation of concerns has been presented using ray tracing as an example. It has been shown that an application can be split into subject-specific, fully encapsulated modules and coordination code that connects them into a streaming network. A coordination language specifically designed for this approach, called S-NET, has been outlined and the programming style it promotes briefly exposed. Experimental evidence has been presented to confirm the viability of coordination using S-NET: not only the example application does not lose performance after recoding it as a coordinated network, in fact due to the asynchronous, message-driven nature of the coordination program its performance exceeds that of the explicit message-passing code in some cases.

The European authors acknowledge EU financial support under grants IST-02761"Æther", IST-215216 "Apple-CORE" and IST-248828 "ADVANCE".

REFERENCES

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

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

[3] G. Stefanescu, *Network Algebra*. Springer-Verlag, 2000.

[4] T. Whitted, "An improved illumination model for shaded display," *Communications of the ACM*, vol. 23, no. 6, pp. 343–349, 1980.

[5] D. Kirk and J. Arvo, "The Ray Tracing Kernel," in *Ausgraph'88, Melbourne, Australia*, 1988, pp. 75–82.

[6] J. Goldsmith and J. Salmon, "Automatic Creation of Object Hierarchies for Ray Tracing," *IEEE Comput. Graph. Appl.*, vol. 7, no. 5, pp. 14–20, 1987.

[7] D. DeMarle, S. Parker, M. Hartner, C. Gribble, and C. Hansen, "Distributed interactive ray tracing for large volume visualization," in *IEEE PVG'03, Seattle, USA*. IEEE Computer Society, 2003.

[8] C. Benthin, I. Wald, M. Scheerbaum, and H. Friedrich, "Ray tracing on the cell processor," in *IEEE RT'06, Salt Lake City, USA*. IEEE Computer Society, 2006, pp. 15–23.

[9] C. Grelck, J. Julku, and F. Penczek, "Distributed S-Net," in *IFL'09, South Orange, NJ, USA*, M. Morazan, Ed. Seton Hall University, 2009.

[10] 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 2.0," University of Hertfordshire, School of Computer Science, Hatfield, England, United Kingdom, Technical Report 499, 2010.

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

[12] A. Shafarenko, "Nondeterministic coordination using s-net," in *High Speed and Large Scale Scientific Computing*, ser. Advances in Parallel Computing, W. Gentzsch, L. Grandinetti, and G. Joubert, Eds. IOS Press, 2009, vol. 18, pp. 74–96.

[13] S. F. Hummel, E. Schonberg, and L. E. Flynn, "Factoring: a method for scheduling parallel loops," *Commun. ACM*, vol. 35, no. 8, pp. 90–101, 1992.

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

[15] E. H. Siegel and E. C. Cooper, "Implementing distributed linda in standard ml," School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, USA, Tech. Rep., 1991.

[16] G. Sutcliffe and J. Pinakis, "Prolog-linda : An embedding of linda in muprolog," Department of Computer Science, The University of Western Australia, Nedlands, 6009, Western Australia, Tech. Rep., 1989.

[17] G. C. Wells, A. G. Chalmers, and P. G. Clayton, "Linda implementations in java for concurrent systems: Research articles," *Concurr. Comput. : Pract. Exper.*, vol. 16, no. 10, pp. 1005–1022, 2004.

[18] G. Florijn, T. Bessamusca, and D. Greefhorst, "Ariadne and HOPLa: flexible coordination of collaborative processes," in *Coordination'96, Cesena, Italy, 15-17 April, 1996. LNCS 1061*, P. Ciancarini and C. Hankin, Eds., 1996, pp. 197–214.

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

[20] G. Michaelson and K. Hammond, "Hume: a functionally-inspired language for safety-critical systems," in *SFP00, University of St Andrews, Scotland, July 26th to 28th, 2000*, ser. Trends in Functional Programming, vol. 2, 2000.

[21] K. Hammond, "Exploiting purely functional programming to obtain bounded resource behaviour: the Hume approach," in *CEFP 2005, Budapest, Hungary, July 4-15, 2005, Revised Selected Lectures*, ser. Lecture Notes in Computer Science, Z. Horváth, Ed., vol. 4164. Springer-Verlag, 2006, pp. 100–134.

[22] R. Loogen, Y. Ortega-Mallén, and R. Peña-Marí, "Parallel functional programming in Eden," *Journal of Functional Programming*, vol. 15, no. 3, pp. 431–475, 2005.

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

[24] R. Ebner and A. Pfaffinger, "Transformation of Functional Programs into Data Flow Graphs Implemented with PVM," in *EuroPVM '96*. London, UK: Springer-Verlag, 1996, pp. 251–258.

[25] L. V. Kale and G. Zheng, "Charm++ and AMPI: Adaptive Runtime Strategies via Migratable Objects," in *Advanced Computational Infrastructures for Parallel and Distributed Applications*, M. Parashar, Ed. Wiley-Interscience, 2009, pp. 265–282.

[26] P. Charles, C. Grothoff, V. A. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, "X10: an object-oriented approach to non-uniform cluster computing," in *OOPSLA'05, San Diego, USA*, R. E. Johnson and R. P. Gabriel, Eds., 2005, pp. 519–538.

[27] B. L. Chamberlain, D. Callahan, and H. P. Zima, "Parallel programmability and the chapel language," *International Journal of High Performance Computing Applications*, vol. 21, pp. 291–312, 2007.

[28] C. E. Leiserson, "The cilk++ concurrency platform," in *46th Design Automation Conference (DAC'09), San Francisco, USA*, 2009, pp. 522–527.

[29] K. Zhang, K. Damevski, and S. Parker, "SCIRun2: A CCA framework for high performance computing," in *HIPS'04, Santa Fé, NM, USA*. IEEE Computer Society, 2004, pp. 72–79.