Cluster Computing as an Assembly Process

Coordination with S-Net

Clemens Grelck
Institute of Informatics
University of Amsterdam,
1098XG Amsterdam, The Netherlands
e-mail: c.grelck@uva.nl

Jukka Julku
VTT Technical Research Centre
P.O.BOX 1000, FI02044, Finland
e-mail: Jukka.Julku@vtt.fi

Frank Penczek, Alex Shafarenko
School of Computer Science
University of Hertfordshire
Hatfield, United Kingdom
e-mail: {f.penczek,a.shafarenko}@herts.ac.uk

***Abstract*— This poster will present a coordination language for distributed computing and will discuss its application to cluster computing. It will introduce a programming technique of cluster computing whereby application components are completely dissociated from the communication/coordination infrastructure (unlike MPI-style message passing), and there is no shared memory either, whether virtual or physical (unlike Open-MP). Cluster computing is thus presented as something that happens as late as the assembly stage: components are integrated into an application using a new form of network glue: Single-Input, Single-Output (SISO) asynchronous, nondeterministic coordination.**

I. THE BIG PICTURE

Consider a streaming network of components, each consuming input streams and producing output streams. This model has been known under the name *stream processing* for many years from as far back as the seminal work of Giles Kahn [1]. It has given rise to a host of programming languages: Lustre[6], Esterel[7], etc. Most results in this area refer to the deterministic, synchronous basic model, which is quite adequate for real-time, distributed control applications, but which lacks flexibility and support for software engineering to be used for general-purpose cluster computing.

The first version of the language S-Net, and the basic principles of asynchronous type-directed stream processing, were proposed by the last author [2,3]. The language was subsequently extended and implemented by the rest of the authors, and recently a cluster-based implementation was produced on top of MPI libraries[4]. The S-Net approach relaxes the constraints of stream processing, which sacrifices real-time guarantees but makes massive gains in compositionality and expressive power.

Specifically:

Cyclic networks are represented as acyclic, infinite (regular) ones. A finite part of this type of network is deployed by progressive unrolling (and can be rolled back when not used). Although not eliminating the possible deadlock, acyclic networks make it much easier to control concurrency and diagnose potential problems.

The topological variety of network nodes is unified by restricting them to Single-Imput-Single-Output (SISO) entities that are supplemented with nondeterministic mergers and type-based splitters to achieve sufficient fan-in and fan-out, respectively. A component inputs a single message via its standard parameter-passing interface and uses a single library function to pass its result messages out (to an unspecified destination).

The logical network structure is encoded using a small set of unary and binary SISO-to-SISO generic *combinators*, which are automatically specialised for message types. As a result, an application assembly can be defined as a hierarchical algebraic formula.

The concept of “type” supports a clear abstraction and encapsulation of user data on the one hand (thus S-Net is a coordination language) and record subtyping and inheritance on the other, thereby allowing for an OOP style program assembly irrespective of the choice of component language. A type inference engine is used as a vehicle of specialisation and configuration, and the programmer is in a position to control the compiler’s automatic choices by providing explicit subtype restrictions and coercions wherever necessary.

An introduction to S-Net can be found in [5]. This poster will present the language outline and, as a main contribution, our thoughts on using S-Net in cluster computing.

II. CLUSTER PROGRAMMING MODEL

The use of S-Net facilitates programming irregular, message driven applications. The proposed methodology is as follows. The parallel aspect of the code is represented in a form that we call Spinal Vector Petri Net (Spinal VPN, or SVPN), which is constructed in the spirit of CPN[8] but is very different from it technically. A VPN conceptualizes an application as a vector of identical graphs. Each graph has places and transitions, similarly to ordinary PNs. A place has a single input arc and a transition has a single output arc, whilst the outputs of a place and inputs of a transition are generally multiple. Tokens have a color and all must carry among other data the net index (NI). Whenever the NI changes, the token is "warped" out to the corresponding graph replica while maintaining its position on the graph. Only places can change token color significantly; transitions always aggregate the contents of all input tokens to produce an output one, without changing any values.

A spinal VPN graph has two dedicated nodes: an input and an output, which are identified with the application's input and output streams. The input and output are connected with a marked path through the graph, called the *spine*. Tokens travelling along the spine are assumed to retain all components of their contents not specifically mentioned on the arcs and in the places (in the spirit of S-Net *flow inheritance* [5]), while the arcs off the spine require all content to be fully specified locally.

The proposed cluster-computing programming method consists in presenting the application in SVPN form and then transforming it (by hand at present, but in future tools will be available) mechanically to an S-Net program as follows:

1. All places are given names and are encoded as S-Net subnets and all transitions are encoded as S-Net synchrocells combined in series with the subnets. S-Net filtration is employed to label each input type of a cell with a binding tag.
2. Using the above and the SVPN graph itself the subnet type signatures are written in S-Net and supplemented with an informal description of the subnet algorithm; the signatures and descriptions together are treated as a specification. Applications programmers are engaged to further refine and implement the subnet code (in S-Net and box languages as appropriate)
3. To obtain a complete application, all subnets are joined in parallel thus:

```
( (S1 || S2 || S3...)!@<ind> )*<out>
```

Here the index tag <ind> corresponds to the NI of the model and at the same time the virtual processor of the S-Net/MPI platform.

The above program has a simple interpretation. The !@ combinator distributes replicas of the net in brackets over the processors of a distributed system, and the star combinator arranges a (conceptually) infinite pipeline with replicas of its argument, which effectively represents iteration time. Data flows along the pipeline and messages that have the tag <out> are extracted into the output stream. The input stream to the whole network delivers the initial data. At any given moment the unfolding of the *-pipeline may not be uniform, some of the !@-replicas could be further down than others. Since synchrocells represent the firing rules for transitions, the flexible asynchronous nature of the computation will be realized to the fullest extent that the data dependencies in the algorithm may allow it.

III. ACKNOWLEDGEMENT AND FUTURE WORK

This work was supported by the EU Framework 6 Integrated Project No 027611 (AETHER), and was a collaborative activity between Universities of Hertfordshire, Amsterdam, Imperial College (London) and VTT Finland. Our experimental tools have been used by *Thales Research and Technology*, Paris to re-code a complex distributed computation of radar scattering data in a cluster-computing style.

This is work in progress. We are currently implementing a distributed, self-balancing Particles-in-Cells code in S-Net/MPI as an example of using the SVPN method for more conventional cluster/grid applications in computational science.

[1] Kahn, G.: The semantics of a simple language for parallel programming. In Rosenfeld, L., ed.: Information Processing 74, Proc. IFIP Congress 74. August 5-10, Stockholm, Sweden, North-Holland (1974) 471–475

[2] Shafarenko, A.: Stream processing on the grid: an array stream transforming language. In: SNPD. (2003) 268–276

[3] A.Shafarenko et al.: Project No FP6-IST-027611. Grant proposal. *April*, 2005

[4] Grelck, C., Julku, J., Penczek, F.: Distributed S-Net. In Morazan, M., ed.: Implementation and Application of Functional Languages, 21st International Symposium, IFL'09, South Orange, NJ, USA, Seton Hall University (2009) 39–54

[5] C. Grelck, S. Scholz, and A. Shafarenko, A Gentle Introduction to S-Net: Typed Stream Processing and Declarative Coordination of Asynchronous Components, Parallel Processing Letters, vol. 18, iss. 2, pp. 221-237, 2008.

[6] The Synchronous Data Flow Programming Language LUSTRE. Nicholas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. Proceedings of the IEEE, Vol 79, No. 9, September 1991

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

[8] K. Jensen. Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Volume 1: Basic Concepts. Springer-Verlag, 1992.