Parallel Program Development for a Recursive Numerical Algorithm: a Case Study

Sergei Gorlatch [*]
Institut für Informatik TU München
Arcisstr. 21, 8000 München 2, F.R.G.
e-mail: gorlatch@informatik.tu-muenchen.de

Abstract

A systematic approach to the parallel program development for a new class of numerical methods on sparse grids is presented. It combines formal design and verification methods based on stream processing functions with simple tools for efficiency evaluation. The use of this approach is demonstrated on a real-life example: two-dimensional integration algorithm.

Keywords: parallel programming, formal design methodology, verification, efficiency evaluation, numerical applications.

1 Introduction

This paper describes the first results of a research project which is carried out with the support from the Alexander von Humboldt Foundation. The work is being done within the research funded by the German Science Foundation under contract SFB 0342 (project A6). The aim is to derive formal methods and programming tools for parallel program specification and development. In our approach we try to achieve the following goals advocated by several authors (e.g. [FT89]):

- to start with a specification which is natural and familiar for the expert in the corresponding problem domain;
- to postpone architecture-dependent design decisions until late in the development process.

[*]The author, a senior research associate at the Institute of Cybernetics (Kiev, the Ukraine), is currently in Munich under the sponsorship of the Alexander von Humboldt Foundation (Germany)

As a methodological framework we use formal methods developed by M. Broy et.al. [Bro89] which are based on stream processing functions. We apply this methodology to the new class of so-called sparse grid numerical algorithms [Zen90]. These algorithms significantly reduce the amount of computations in comparison with conventional grid algorithms and are recursive in nature.

The development of parallel programs from original mathematical specifications is a very promising and actively investigated area. There exist not only formal development methods (they are surveyed in [KLGG89]) but also supporting systems such as Model [TSSP85], Crystal [CCL89] and Suspense [RW89]. However these methods and tools do not work in the problem domain of sparse grids because of the inherently recursive and non-homogeneous nature of the corresponding numerical algorithms.

The central point in our approach is the so-called abstract (data-flow) implementation of a specification. For representing an abstract implementation the Applicative Language (AL) is used: it allows us to express streams, stream processing agents (processes) and networks of agents. The abstract implementation of algorithms in AL can be verified with respect to the specification using the denotational semantics of the language [BDD$^+$92a]. Programs in the AL language correspond quite directly to nets of loosely-coupled asynchronously communicating agents. Such nets were investigated, e.g., in [Den85] and used in the languages Lucid [WA85] and Sisal [FCO90]. AL-programs play a key role in our approach: on the one hand, they contain all the potential parallelism of the original numerical specification, on the other hand, a transformation and verification formalism is developed for them, moreover, they can be implemented and their efficiency can be estimated. The transformation of the AL-programs into parallel programs for particular multiprocessor architectures and corresponding experiments constitute the future stage of our project.

In this paper we describe the modification of the general design methodology [BDD$^+$92a] for the case of numerical recursively defined algorithms. The presentation is illustrated by the systematic development of two AL-implementations (a straightforward one and an optimized one) for the algorithm of two-dimensional integration on a sparse grid. Technical details are taken out into Appendix. We also present a simplified variant of an abstract implementation that corresponds to the functional program considered in [Zen92]. Modifications for the case of adaptive integration algorithm are briefly analyzed. Then a simple method of efficiency evaluation for abstract programs is outlined. In the conclusion, the future research is discussed.

In this case study we deliberately do not give a completely formal presentation, but try to keep the derivation readable and understandable. In particular, formal proofs are omitted.

2 Methodology overview

In this section we firstly present a short overview of the general design methodology Focus [Bro89], [BDD$^+$92a] and then outline its modification for the numerical problems taken into consideration. We mainly aim at providing an initial understanding, therefore all complications are avoided.

Focus uses a descriptive functional approach to the development of distributed systems. The development is organised as a sequence of steps. Basically we have the following four levels of system development: requirements specification, design specification, abstract implementation and concrete implementation. In the course of development the description of a system is transformed and refined. At each step the description is verified with respect to the previous one. The basic modeling notion used are sequences of elements called streams: streams of actions (traces) and streams of messages.

The Focus methodology was used up to now mostly for the development of distributed systems interacting with an environment by receiving and producing streams of messages [BDD$^+$92b]. Here we try to modify this methodology having in mind its use for a multiprocessor implementation of a certain class of numerical algorithms. In particular, the "meaning" of some development levels is changed in comparison with the original methodology.

The main feature of the problems we are interested in is that they usually have a precise mathematical specification. We will call it a *requirement specification*. This specification does not have to be constructive (e.g.: "compute an integral for the function f in a given domain with a given accuracy"). To find corresponding algorithmic concepts and investigate their adequacy is the task for experts in numerical methods. They develop a constructive representation which we will consider as a *design specification*. We consider the design specification as the entry point in our approach. Such a specification is usually still mathematical (e.g.: a system of recursive equations determining relations between the matrix of coefficients, the vector of the right hand side and the solution vector in the Gauss method for linear systems solution). Our aim is to transform such a design specification into a parallel program.

As an intermediate level we develop a so-called abstract implementation which is represented in a particular applicative language called AL. The AL-programs are still quite abstract but already executable and thus are called *abstract programs*. An abstract program describes a net of concurrently working agents which asynchronously exchange messages over unbounded directed channels. A program consists of a number of agent declarations and a system of equations describing their interconnection.

The relation between the design specification and its abstract implementation is provided by the denotational semantics of the AL language which, together with transformation rules for the language, is developed by F. Dederichs in his Ph.D. thesis. The semantics assigns a set of stream processing functions to every agent declaration. In this semantical framework the implementation and the equivalence relations are formalized: an agent definition O implements an agent definition I if the set of stream processing functions constituting the semantics of O is a subset of the set of functions constituting the semantics of I. Agents are equivalent if both sets coincide. Using these relations we can formally verify different variants of AL–programs implementing a design specification.

An abstract program should then be transformed into a *concrete program* for the target multiprocessor. This final step, however, is not in the scope of this paper.

3 Algorithms on sparse grids

We outline here the general idea of sparse grid algorithms and present the example (design) specification which is considered in the following.

Grids are called "sparse" because of their analogy to sparse matrices. For two-dimensional problems on the unit square with the degree of partition m (i.e. the boundary meshwidth 2^{-m}) the associated sparse grids contain only $O(m \log m)$ grid points instead of $O(m^2)$ for the usual "full" grids (see Fig. 1).

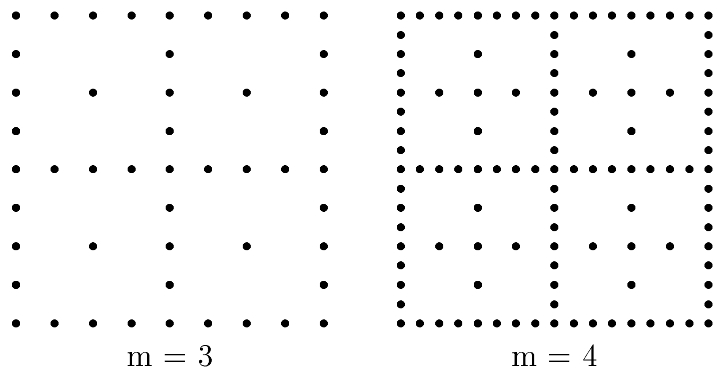

Figure 1: Points in the square sparse grid for m=3 and m=4

It can be shown (see [Zen90]) that sufficiently smooth functions are represented on sparse grids with nearly the same accuracy as on full grids. Thus, the main advantage of sparse grids is that the dimension of the used space (and the amount of necessary computations) is reduced significantly, whereas the accuracy of approximation deteriorates only slightly. The same idea works even better in the multidimensional case and was successfully used for a variety of numerical grid methods ([Gri90],[Zen90]).

As a simple example of a sparse grid method we consider an algorithm for numerical two-dimensional integration. It is developed and implemented on a conventional computer by Th. Bonk at the Technical University of Munich. To simplify the presentation, we restrict ourselves at first to the non-adaptive version of the algorithm, which uses the meshwidth value m as a parameter. In the adaptive case the grid is built dynamically for the required accuracy and thus may be heterogeneous. The idea of the algorithm goes back to Archimedes and is based on domain partition.

The value of the integral for a given function f vanishing on the boundary in the domain $[a1, b1] \times [a2, b2]$:

$$q = \int_{a1}^{b1} \int_{a2}^{b2} f(x1, x2) dx1 dx2$$

is computed for the given meshwidth 2^{-m} as q $= A(a1, b1, a2, b2, m)$ where the function A is defined recursively using the auxiliary functions N and HB as follows:

$$\left.\begin{array}{rcl} A(a1,b1,a2,b2,m) & = & \text{if } m=0 \text{ then } 0 \text{ else } A(a1,\frac{a1+b1}{2},a2,b2,m-1)+ \\ & & A(\frac{a1+b1}{2},b1,a2,b2,m-1)+N(a1,b1,a2,b2,m) \\ N(a1,b1,a2,b2,m) & = & \text{if } m=0 \text{ then } 0 \text{ else } N(a1,b1,a2,\frac{a2+b2}{2},m-1)+ \\ & & N(a1,b1,\frac{a2+b2}{2},b2,m-1))+HB(a1,b1,a2,b2) \\ HB(a1,b1,a2,b2) & = & Expr\{f(a1,a2),f(a1,b2),f(b1,a2),f(b1,b2), \\ & & f(\frac{a1+b1}{2},a2),f(\frac{a1+b1}{2},b2),f(a1,\frac{a2+b2}{2}),f(b1,\frac{a2+b2}{2}), \\ & & f(\frac{a1+b1}{2},\frac{a2+b2}{2}),a1,b1,a2,b2\} \end{array}\right\}(1)$$

For simplicity we use the informal notation $Expr$ reflecting just the values it depends on, rather than the precise expression for the function HB.

For specifications similar to (1) a strict computational semantics based on fixed-point theory can be constructed as in [PZ81]. This semantics can be used for proving correctness of an implementation with respect to the corresponding specification.

4 Abstract program development

In this section we describe the construction of an abstract program from the design specification (1). An abstract program in the AL language consists of a number of agent declarations and a system of equations describing their interconnections.

4.1 Straightforward implementation

The first variant of an abstract implementation corresponds quite directly to classical data-flow programs [Den85]. The corresponding data-flow graph (we will call it a net in the sequel) is presented in figure 2.

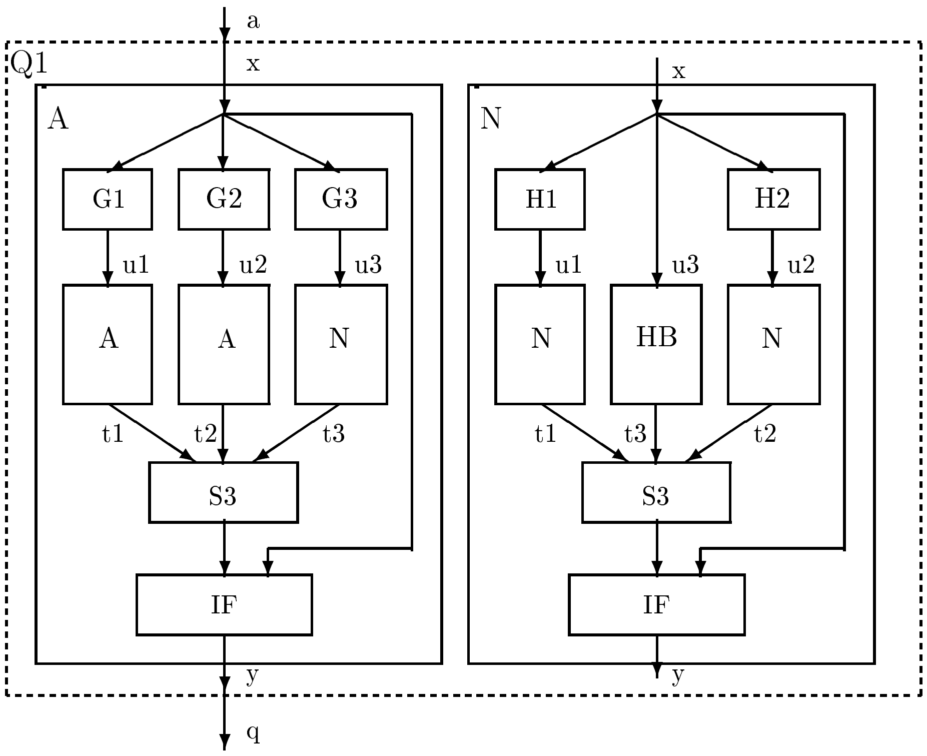

Figure 2: Data-flow net for the specification

This net is derived straightforwardly: an agent is introduced for every operation or function call in the original specification.

We use three agents implementing the functions A, N and HB; these agents are also denoted by A, N and HB. Moreover the following auxiliary agents are used:

IF - implements a conditional expression,

S3 - sums up three real values,

G1, G2, G3 - compute arguments for the recursive call of A,

H1, H2 - compute arguments for the recursive call of N.

We consider S3 and HB as elementary agents, although it would be possible to represent them as combinations of simpler agents.

A data-flow net can directly be represented by a corresponding AL-program. All variables and channels in a program belong to some type. We will use the basic types `int` and `real`. For handling tuples of values we also define a new type using standard techniques of abstract data types. Type `r4i` defines a tuple of four real values and one integer value. We define the operation [, , , ,] of tuple construction and the operation .i that gives the i-th element of a tuple.

In our case we have only one-element streams (this element may be a tuple). This allows us to simplify the AL syntax in comparison with the general case making no difference between streams and stream elements.

The AL-program derived from the specification (1) reads as follows:

```
program Q1 = chan r4i a → chan real q :
   agent A = chan r4i x → chan real y :
      y = IF(x, s); s = S3(t1, t2, t3);
      t1 = A(u1); t2 = A(u2); t3 = N(u3);
      u1 = G1(x); u2 = G2(x); u3 = G3(x);
   end;
   agent N = chan r4i x → chan real y:
      y = IF(x, s); s = S3(t1, t2, t3);
      t1 = N(u1); t2 = N(u2); t3 = HB(x);
      u1 = H1(x); u2 = H2(x);
   end;
   agent HB = chan r4i x → chan real y:
      y = Expr{f(x.1,x.3), f(x.1,x.4), f(x.2,x.3), f(x.2,x.4), f((x.1+x.2)/2,x.3),
                f((x.1+x.2)/2,x.4), f(x.1,(x.3+x.4)/2), f(x.2,(x.3+x.4)/2),
                f((x.1+x.2)/2,(x.3+x.4)/2), x.1, x.2, x.3, x.4};
   end;
   agent S3 = chan real x, y, z → chan real w: w = x+y+z; end;
   agent G1 = chan r4i x → chan r4i y: y = [x.1,(x.1+x.2)/2,x.3,x.4,x.5-1]; end;
   agent G2 = chan r4i x → chan r4i y: y = [(x.1+x.2)/2, x.2, x.3, x.4, x.5-1]; end;
   agent G3 = chan r4i x → chan r4i y: y = [x.1, x.2, x.3, x.4, x.5-1]; end;
   agent H1 = chan r4i x → chan r4i y: y = [x.1, x.2, x.3, (x.3+x.4)/2, x.5-1]; end;
   agent H2 = chan r4i x → chan r4i y: y = [x.1, x.2, (x.3+x.4)/2, x.4, x.5-1]; end;
   agent IF = chan r4i x, chan real y → chan real z:
      z = if x.5 = 0 then 0 else y fi;
   end;
q = A(a)
end
```

First the program name and its input and output streams are defined. The program Q1 receives a tuple [a1,b1,a2,b2,m] as input and produces a real value q as output. Then follow the agents declarations and the equational part of the program (here it consists of just one equation). In the headers of the program and agents, streams are declared by means of the keyword chan. The agents have zero or more named input parameters and one or more named output parameters. The body of an agent is built by equations just like the equational part of a complete program. Every equation has a number of stream identifiers on the left hand side and an expression of adequate arity and type on the right hand side. On the left hand side an output stream occurs exactly once while input streams may occur only on the right hand side. Streams that are neither inputs nor outputs are called internal (e.g. t1 is an internal stream). We again use the abbreviation *Expr* in the agent HB for simplicity.

An agent may be called in its own body (see agents A and N). This corresponds to the so-called recursion in place which means that agents can be unfolded thereby leading to a number of different instantiations of the same agent working in parallel. Each A- or N-instantiation in Q1 can begin its work independently of the outputs of other instances. Therefore the generation of new instantiations is restricted only by the time used for preparing their input streams in the agents G1, G2, G3, H1, H2.

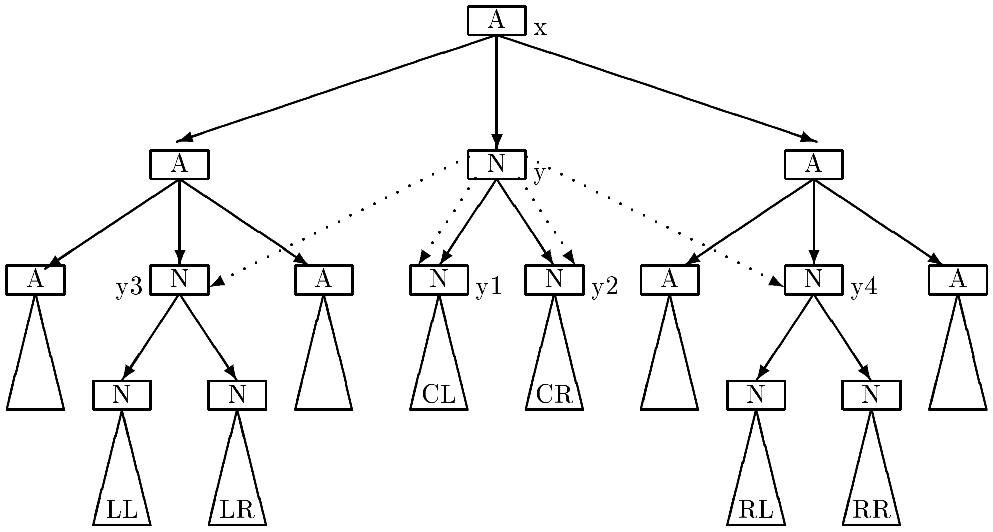

Figure 3: A fragment of the recursion tree

Consider the tree of recursive calls for the program Q1 where any node corresponds to one instantiation of agents A or N (we call them A- and N-instantiations) and an edge from one node to another means that the latter instantiation is called by the first one. Nodes are labeled by the names of corresponding instantiations and are called A- or N-nodes. A fragment of this tree is illustrated in figure 3 by solid lines. Subtrees are depicted as triangles. The meaning of dotted lines and the names of subtrees will be explained in the sequel. Assume that the root of the fragment - the node x - corresponds to the instantiation A(a,b,c,d,i) and thus has the recursion level i (the root of the whole tree has level m and all the leaves have level 0).

The process of generating new agent instantiations in the program Q1 finishes when the current value of the recursion level decreases to zero. Each agent begins to work as soon as the necessary input values are computed (by other agents) and received via its input streams. E.g. agent IF can start to work without a value of its second input provided the fifth component of the first input is equal to zero. In this case it produces zero as output. Instantiations in the program Q1 finish their work in an order opposite to the order of their generation, because every instantiation depends on the output of the instantiation called from it.

In the sequel we present some facts which are supposed to be true but are not proved formally. We call them claims.

Claim 1. The program Q1 implements the specification (1).

The proof should be based on the formal semantics for specifications and AL-programs and on the definition of the implementation relation (see Section 2).

4.2 Optimization: avoiding repetitive computations

An abstract program that is directly derived from the design specification may of course be inefficient in some respect.

One possible optimization at the level of abstract programs is to eliminate repetitive computations. In our example all necessary values of the function f are computed in each instantiation of the agent N (more precisely, in the agent HB within the corresponding instantiation of N). In the sequel we analyze which particular values are computed in more than one instantiation and then develop a new version of an abstract program in which all the repetitive computations are eliminated.

Consider the computations in the N-nodes shown in figure 3. The node y corresponds to the instantiation N(a,b,c,d,i), which computes HB in the rectangle [a,b]×[c,d], and the instantiations of the level i-1: y1, y2, y3, y4 compute HB in the corresponding halves of this rectangle (see figure 4).

To perform these computations, every instantiation needs values of f in nine points of the corresponding rectangle. We will use straightforward abbreviations for these values: "z" for the central point of the rectangle, "nw" for the north-west point, "se" for the south-east and so on. Thus for the instantiation N(a,b,c,d,i), $0 \leq i \leq m$ we have: nw = f(a,d), n = f((a+b)/2,d), ne = f(b,d), e = f(b,(c+d)/2), se = f(b,c), s = f((a+b)/2,c), sw = f(a,c), w = f(a,(c+d)/2), z = f((a+b)/2,(c+d)/2). These abbreviations will be used together with the name of the corresponding node in the recursion tree, e.g. nw(x) denotes the north-west point for node x, [nw,ne](y) denotes the tuple of north-west and north-east values for node y, etc.

From figure 4 we can see that the instantiations of the levels i and i-1 use some common values. More presicely, this is expressed in the following proposition where si.w denotes the i-th son of the node w.

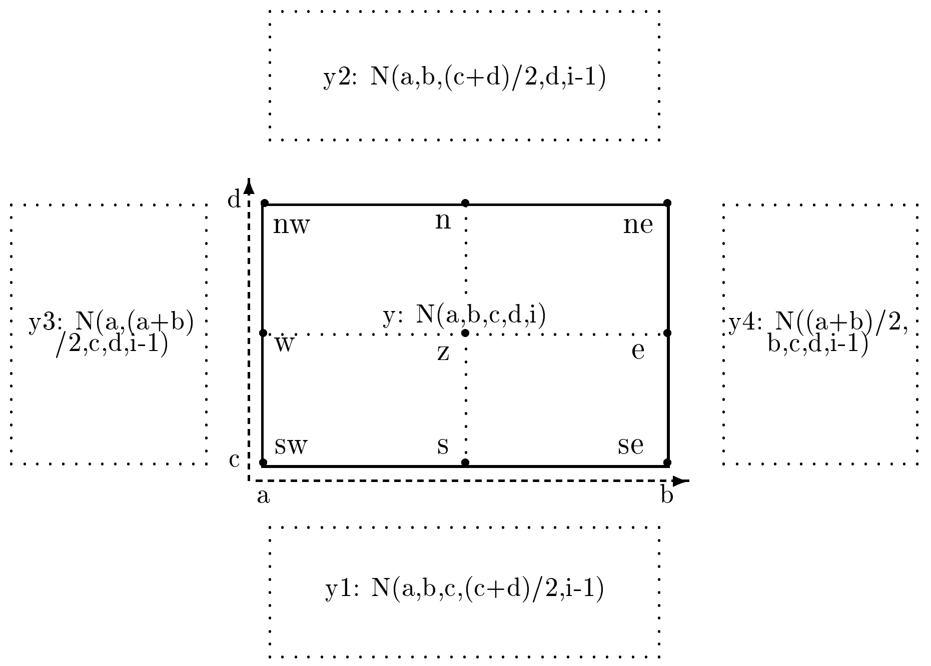

Figure 4: Computation of N

Proposition 1. For every A-node x of non-zero level the following N-nodes : y=s2.x, y1=s1.y, y2=s2.y, y3=s2.s1.x, y4=s2.s3.x have the property:
[nw,n,ne,se,s,sw](y1) = [w,z,e,se,s,sw](y), [nw,n,ne,se,s,sw](y2) = [nw,n,ne,e,z,w](y), [nw,ne,e,se,sw,w](y3) = [nw,n,z,s,sw,w](y), [nw,ne,e,se,sw,w](y4) = [n,ne,e,se,s,z](y).

The nodes $y, y1, y2, y3, y4$ can be seen in figure 3. In contrast to the claims, the proofs of propositions are known to the author but are omitted here for brevity.

We will distinguish two classes of N-instantiations: those called from A (N1-instantiations) and those called from N (N2-instantiations). The notion "N-instantia tion" is used for instantiations from both classes. The N1- and N2-nodes in the recursion tree are treated analogously.

Proposition 2. For every N2-node w of non-zero level holds:

$$[\text{nw,n,ne,se,s,sw}](\mathsf{s1}.w) = [\text{w,z,e,se,s,sw}](w),$$
$$[\text{nw,n,ne,se,s,sw}](\mathsf{s2}.w) = [\text{nw,n,ne,e,z,w}](w).$$

It follows from the propositions 1 and 2 that every N-instantiation except that of the level m, uses 6 values which are already computed in the N-instantiations of the previous levels. Therefore we can avoid repetitive computations by means of additional communications between N-instantiations (the corresponding communication lines between levels i and $i-1$ are shown in figure 3 by dotted lines).

Consider now another source of repetitive computations in the program Q1. We will use the following functions for an arbitrary node w of the recursion tree: tree.w - subtree with w as a root, tree*.w - the same subtree without leaves. The left and the right subtrees of the binary tree T are denoted left.T and right.T, correspondingly.

For the A-node x in figure 3 consider the following subtrees of the recursion tree:

C(x) = tree*.s2.x, L(x) = tree.s2.s1.x, R(x) = tree.s2.s3.x

and their subtrees:

CL(x) = left.C(x), CR(x) = right.C(x), LL(x) = left.L(x),

LR(x) = right.L(x), RL(x) = left.R(x), RR(x) = right.R(x)

(some of these subtrees can be seen in figure 3). Here C is used as an abbreviation for "center" , L for "left" and R for "right".

Proposition 3. For every A-node x of the level more than 1, the trees CL(x), CR(x), LL(x), LR(x), RL(x), RR(x) are isomorphic to each other and for any collection cl, cr, ll, lr, rl, rr of the corresponding isomorphic nodes from these trees, the following property holds:

[w,e](ll) = [w,z](cl), [w,e](rl) = [z,e](cl), [w,e](lr) = [w,z](cr), [w,e](rr) = [z,e](cr).

All the nodes in the subtrees from proposition 4 correspond to N2-instantiations. Therefore, all N2-nodes except those contained in the subtree tree.s2.r (r is the root of the whole recursion tree), use 2 values, namely w and e which are already computed in the N2-instantiations of the previous levels. These values would not be computed repetitively if we introduce some additional communications.

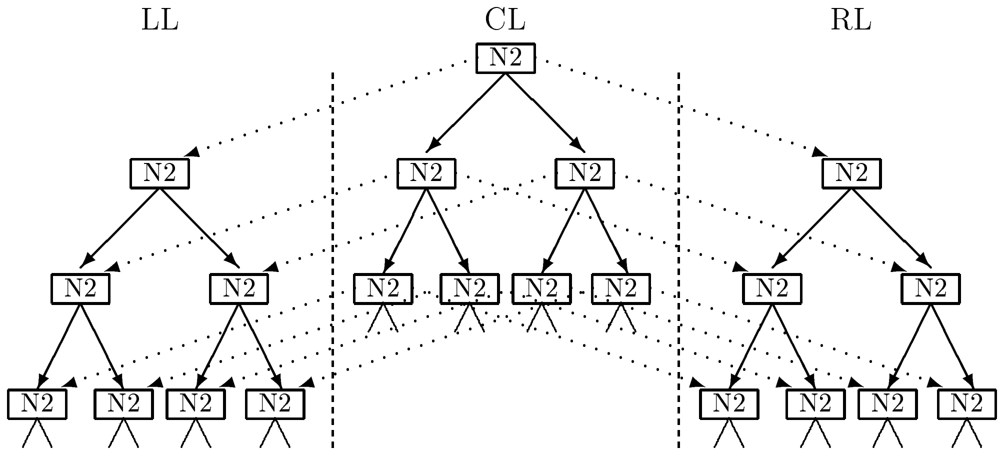

Figure 5: Communications between N2-instances

In figure 5 the necessary communications between N2-instantiations in the subtrees CL, LL, RL of the recursion tree are shown by dotted lines.

The method of program optimization follows from propositions 1 - 3. Corresponding considerations are presented in appendix as they are quite space consuming. We present here only the data-flow net for the optimized program Q2 (see figure 6).

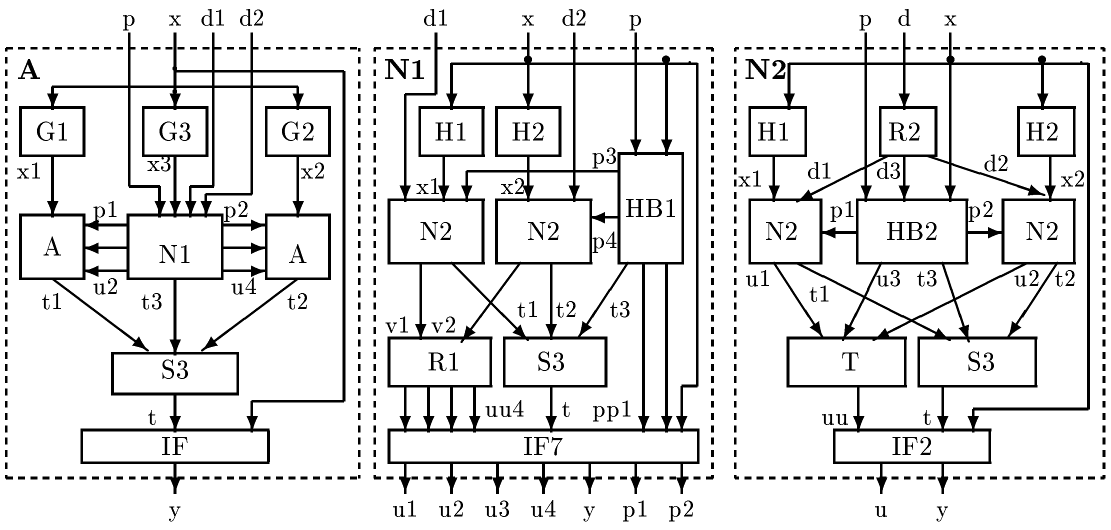

Figure 6: The program Q2

4.3 Adaptive algorithm

We make here some comments about an adaptive version of the integration algorithm. The specification for the adaptive case reads as follows:

$$\left.\begin{aligned}
A(a1,b1,a2,b2,\varepsilon) &= \text{if } HB(a1,b1,a2,b2) < \varepsilon \text{ then } N(a1,b1,a2,b2,\varepsilon)\\
&\quad \text{else } A(a1,\tfrac{a1+b1}{2},a2,b2,m-1)+\\
&\quad A(\tfrac{a1+b1}{2},b1,a2,b2,m-1)+N(a1,b1,a2,b2,m)\\
N(a1,b1,a2,b2,\varepsilon) &= \text{if } HB(a1,b1,a2,b2) < \varepsilon \text{ then } HB(a1,b1,a2,b2)\\
&\quad \text{else } N(a1,b1,a2,\tfrac{a2+b2}{2},\varepsilon)+\\
&\quad N(a1,b1,\tfrac{a2+b2}{2},b2,\varepsilon)+HB(a1,b1,a2,b2)\\
HB(a1,b1,a2,b2) &= Expr\{f(a1,a2),f(a1,b2),f(b1,a2),f(b1,b2),\\
&\quad f(\tfrac{a1+b1}{2},a2),f(\tfrac{a1+b1}{2},b2),f(a1,\tfrac{a2+b2}{2}),f(b1,\tfrac{a2+b2}{2}),\\
&\quad f(\tfrac{a1+b1}{2},\tfrac{a2+b2}{2}),a1,b1,a2,b2\}
\end{aligned}\right\}(2)$$

Here ε denotes the required accuracy of the solution. The programs Q1 and Q2 can be easily transformed into adaptive versions. However, the degree of parallelism in Q1 will be reduced: new A-agents can be called only after evaluating the corresponding value of the HB function. The program Q3 can be also made adaptive as in [Zen92] but only with the assumption that all necessary boundary values of the function f for the given ε. are computed beforehand.

5 Efficiency evaluation

In this section we advocate the use of special tools for predicting the efficiency of abstract programs and outline one possible simulation method for our case study.

Having two functionally equivalent versions of an abstract implementation we would like to be able to compare their efficiency. For our example we are interested particularly in the following. Agents in the program Q2 perform less computations than in Q1, but more communications and data dependencies are involved. Thus the number of agents working in parallel in Q2 can decrease. We should analyse therefore which factor prevails in affecting the efficiency.

The efficiency evaluation at the applicative level is also very important because of the optimization problem. For applicative languages, especially for those intended as AL for parallel implementation, optimization is a necessity and the potential gain can be by an order of magnitude.

The straightforward way to evaluate program efficiency is to execute the program on the target machine. In this case however the result is substantially determined by the features of the target machine and by the peculiarities of the implementation. One has to take into account costs for computation and communication, interconnection topology, OS characteristics and so on. Moreover, the experimentation with large parallel systems is often quite a complex and an expensive job.

We therefore advocate the use of special simulation tools on a conventional computer. We raise the following claims for these tools. Firstly, they must be flexible enough to allow program simulation in various multiprocessor configurations. Secondly, simulated parallel programs must be relatively simple, otherwise it would require too much time to investigate their efficiency on a serial computer.

We are able to satisfy these requirements using the following observations:

- at the stage of abstract implementation we are mostly not interested in receiving the exact numerical result produced by the program but in obtaining information about its so-called control behaviour: parallel processes interleaving, communications, waiting times etc.; therefore, we can neglect the "computational" part of the program;

- numerical programs we are interested in have the feature that their control behaviour usually depends on a very small number of parameters. In particular, for our example the number of agents working in parallel depends on the parameter m and is independent of $a1, b1, a2, b2$ and the function f. This allows to simplify the simulation.

A simulation method can be described briefly as follows. An AL-program which is to be simulated is transformed into a special simplified version called timed-control model. This model reflects only those aspects of the original program that affect its control behaviour. The corresponding objects and streams (as m in Q1) are called control data, and for them simulation is identical to the real execution: their values are computed and transmitted through channels. For all other objects and streams we simulate only timing and synchronization.

An elementary time unit is arbitrarily chosen and all time values are expressed based on this unit. We assign an "execution time value" to every agent containing no calls of other agents or recursive calls, e.g. among all elementary agents in Q1 the most time-consuming one is surely HB. "Time value" for a channel reflects the time required for transmitting one stream element through this channel. All time values can be expressed using parameters; choosing appropriate parameters values we can tune the model towards the target machine with particular characteristics.

As a result of simulation we obtain a timed history of the program execution including communications between agents, waitings etc.

The simulation method described is very simple and can give only a preliminary idea and information about the control behaviour of an abstract program. These results however can be of great importance at the initial stages of parallel program development.

6 Conclusion and future work

In the paper we have shown how a systematic methodology can be used for parallel program development in the particular problem domain of sparse grid algorithms.

This small case study induced us to take into consideration very different aspects of parallel computing: formal specification and verification, design methodology, efficiency evaluation and simulation, numerical methods. The research carried out was useful for understanding the advantages and weak points of the approach.

We see the novel feature of our development approach in using an abstract implementation level that allows us to achieve the following:

- the design specification is formulated at a level of abstraction which is very close to mathematical description; specification development thus can be left to the expert user;

- the development and transformation of the abstract implementation can be formally verified ("provably correct design");
- the efficiency of implementation decisions can be evaluated at the level of abstract programs by means of simulation;
- the abstract implementation is architecture-independent and can be mapped onto various parallel architectures.

These benefits are surely not easily gained. In the future we are going to develop the approach described here in the following directions: we will

- derive simplified verification and transformation techniques for abstract programs using problem domain knowledge; e.g. this techniques should make the proofs for the claims 1 and 2 easier;
- experiment with real multiprocessors (e.g., Hypercube-architecture) in order to investigate various opportunities of parallel implementation of AL-constructs;
- implement the described simulation method on a conventional computer.

One of the main problems concerning implementation of AL-programs is how to make the process grain size (execution time between communication operations) large enough to permit efficient use of a machine that has comparatively high communication costs (e.g. Hypercube). We can not simply create a process for each potentially parallel agent: this could often lead to programs whose execution is slower than the sequential one. Grain size of a process can be increased by merging agents that interact only with each other. Quite a simple solution exists for our example algorithm: all newly created agents instantiations (processes) are mapped onto free processors as long as those are available; then the whole "tree" of processes, recursively created from the given one, remains on the same processor working in a multiprogramming mode.

7 Acknowledgements

The author is very grateful to the Alexander von Humboldt Foundation whose generous financial and organizational support made this project a reality.

I should stress the importance for this work of the experience gained within the Kiev macroconveyor project under the guidance of Prof. Yulia V. Kapitonova and Prof. Alexander A. Letichevsky.

I am very much obliged to my academic host in Germany - Prof. Manfred Broy for his kind encouragements. I am indebted to Frank Dederichs, Max Fuchs and

Dr. Ketil Stoelen for helpful discussions on this paper. The area of sparse grid algorithms was brought to my attention by Prof. Christoph Zenger and was several times discussed with Thomas Bonk. Many thanks are due to Dr. Thomas Bemmerl for his help on the Hypercube software.

8 Appendix

This part of the paper includes the following three sections:

1. implementation of additional communications which are necessary for avoiding repetitive computations;
2. text of the optimized program Q2;
3. simplified version of the optimized program that corresponds to the considerations in [Zen92].

8.1 Additional Communications

To implement the communications described in section 4.2 we will use one-element streams of the type $\mathsf{r6}$. We assume that every N-instantiation has an additional input stream p; moreover, every N1-instantiation has four output streams $p1, p2, p3, p4$ and every N2-instantiation has two output streams $p3, p4$.

Output streams for every N-instantiation y are defined as follows:
$p1(y) = [\text{nw, n, z, s, sw, w}](y)$, $p2(y) = [\text{n, ne, e, se, s, z}](y)$,
$p3(y) = [\text{nw, n, ne, e, z, w}](y)$, $p4(y) = [\text{w, z, e, se, s, sw}](y)$.

The value of the input stream for the given N-node is defined by the output stream of the node from the previous level as follows:

1) for every A-node x of non-zero level:
$\text{p}(\mathsf{s2.s1}.x) = p1(\mathsf{s2}.x)$, $\text{p}(\mathsf{s2.s3}.x) = p2(\mathsf{s2}.x)$;

2) for every N-node y of non-zero level: $p(\mathsf{s1}.y) = p3(y)$, $p(\mathsf{s2}.y) = p4(y)$;

3) if y is the N-node of the level m then $p(y) = \varepsilon$.

Proposition 4. For every N1-node y of a level less than m holds:
$[\text{nw, ne, e, se, sw, w}](y) = p(y)$,
and for every N2-node y holds:
$[\text{nw, n, ne, se, s, sw}](y) = p(y)$.

The proof follows directly from the propositions 1 and 2 and the definition of input and output streams.

Proposition 4 makes it possible that in every N-instantiation (except one of the level m) the corresponding 6 values may be received from the input stream and thus are not computed repetitively.

To implement necessary communications for managing the second source of repetitive computations (see proposition 3 and figure 5) we use streams organized as binary trees of tuples. The corresponding abstract data type $\mathsf{tree}(\tau)$ (τ is a type of tuples in the nodes) is assumed to allow the following operations: $\mathsf{head}.t$ - gives the root tuple of the tree t, $\mathsf{left}.t$ and $\mathsf{right}.t$ give corresponding subtrees of t. We will distinguish two kinds of streams: input and output.

Every N1-node y has four output streams: $U1(y), U2(y), U3(y), U4(y)$. They are built as trees isomorphic to $LL(x), LR(x), RL(x), RR(x)$ correspondingly and they are therefore isomorphic also to $CL(x), CR(x)$ where $x = \mathsf{fa}.y$ (father of y). Elements of these streams are defined as follows:

$u1$ = [w,z](cl), $u2$ = [w,z](cr), $u3$ = [z,e](cl), $u4$ = [z,e](cr),

where $(u1, u2, u3, u4, cl, cr)$ is an arbitrary collection of isomorphic nodes from $U1, U2, U3, U4, CL, CR$ correspondingly.

For any N2-node w we define N1-node $\mathsf{host}.w$ such that $w \in \mathsf{tree}.\mathsf{host}.w$. From the definition of output streams follows that every N2-instantiation w of non-zero level produces one element for each of two corresponding output streams of $\mathsf{host}.w$; these elements are: [w,z](w) and [z,e](w).

On the other hand, with every N1-node y of non-zero level we associate two input streams $D1(y), D2(y)$ which are defined using output streams for the node of the previous recursion level. For every A-node x of the non-zero level and the corresponding $y = \mathsf{s2}.x$ we assume:

$D1(\mathsf{s2}.\mathsf{s1}.x) = U1(y)$, $D2(\mathsf{s2}.\mathsf{s1}.x) = U2(y)$,

$D1(\mathsf{s2}.\mathsf{s3}.x) = U3(y)$, $D2(\mathsf{s2}.\mathsf{s3}.x) = U4(y)$.

Input streams for N1-node of the level m are assumed to be empty.

From proposition 3 and the definitions of streams we derive the following:

Proposition 5. For every N1-node y of the level less than m and any collection of isomorphic nodes $cl, cr, d1, d2$ from $CL(\mathsf{fa}.y), CR(\mathsf{fa}.y), D1(y), D2(y)$ correspondingly, holds: [w,e]$(cl) = d1$, [w,e]$(cr) = d2$.

From proposition 5 follows that every N2-instance w except those contained in the subtree $\mathsf{tree}.\mathsf{s2}.r$ can receive the tuple of values [w,e](w) from the corresponding input stream of the node $\mathsf{host}.w$ and thus must not compute them itself.

8.2 Optimized implementation

We use the propositions 1 – 5 in order to develop a new version of an abstract implementation for the specification (1), namely the program Q2, in which all repetitive computations are eliminated. The names of channels in Q2 correspond almost directly to the names of the streams introduced above.

Some comments, however, seem to be useful for understanding the program Q2:

- instead of agent N we have two agents N1 and N2 corresponding to the classes of instantiations introduced above;
- all streams are sent to the agent N1 through its father A-agent; therefore the agent A in addition to the channel x has also input channels $p, d1, d2$ which correspond to the streams $p, D1, D2$; for the first instantiation of the agent these streams are empty;
- output channels of the agent N1: $u1, u2, u3, u4, p1, p2$ implement the corresponding streams; the streams $p3, p4$ are implemented by inner channels with the same names;
- we use different variants of the agent HB for the agents N1 and N2 because from the proposition 5 the 2-tuples are produced and used only in the N2-instantiations; moreover, N1 and N2 differ also in producing 6-tuples;
- as the streams $u1$ and $u3$ have a common value z in every pair of corresponding nodes, they are formed in the agent T as one stream with 3-tuples of the form [w,z,e]; the same is done for $u2$ and $u4$; these two streams of 3-tuples are then decomposed by the agent R1 into four streams with 2-tuples.

The program Q2 reads as follows (see also figure 6).

```
program Q2 = chan r4i a → chan real q :
  agent A = chan r4i x, chan r6 p, chan tree(r2) d1,d2 → chan real y :
    y = IF(x, t); t = S3(t1, t2, t3);
    t1 = A(x1, p1, u1, u2); t2 = A(x2, p2, u3, u4);
    (t3, p1, p2, u1, u2, u3, u4) = N1(x3, p, d1, d2);
    x1 = G1(x); x2 = G2(x); x3 = G3(x);
  end;
  agent N1 = chan r4i x, chan r6 p, chan tree(r2) d1,d2→
                              chan real y,chan r6 p1, p2, chan tree(r2) u1, u2, u3, u4 :
    (y, p1, p2, u1, u2, u3, u4) = IF7(x, t, pp1, pp2, uu1, uu2, uu3, uu4);
    t = S3(t1, t2, t3); (uu1, uu2, uu3, uu4) = R1(v1, v2);
    (t1, v1) = N2(x1, p3, d1); (t2, v2) = N2(x2, p4, d2);
    (t3, pp1, pp2, p3, p4) = HB1(x, p); x1 = H1(x); x2 = H2(x);
  end;
  agent N2 = chan r4i x, chan r6 p, chan tree(r2) d → chan real y, chan tree(r3) u :
    (y, u) = IF2(x, t, uu); t = S3(t1, t2, t3); uu = T(u1, u2, u3);
    (t1, u1) = N2(x1, p1, d1); (t2, u2) = N2(x2, p2, d2);
    (t3, p1, p2, u3) = HB2(x, p, d3);
    (d1, d2, d3) = R2(d); x1 = H1(x); x2 = H2(x);
```

```
end;
agent R1 = chan tree(r3) v1, v2 → chan tree(r2) u1, u2, u3, u4 :
    head.u1 = [head.v1.1, head.v1.2]; head.u2 = [head.v2.1, head.v2.2];
    head.u3 = [head.v1.2, head.v1.3]; head.u4 = [head.v2.2, head.v2.3];
    (left.u1, left.u2, left.u3, left.u4) = R1(left.v1, left.v2);
    (right.u1, right.u2, right.u3, right.u4) = R1(right.v1, right.v2);
end;
agent HB1 = chan r4i x, chan r6 p → chan real t,chan r6 p1, p2, p3, p4 :
    t = Expr{sw, nw, se, ne, s, n, w, e, z, x.1, x.2, x.3, x.4};
    p1 = [nw,n,z,s,sw,w,]; p2 = [n,ne,e,se,s,z];
    p3 = [nw,n,ne,e,z,w]; p4 = [w,z,e,se,s,sw];
    nw = if p=ε then f(x.1,x.4) else p.1;
    ne = if p=ε then f(x.2,x.4) else p.2;
    e = if p=ε then f(x.2,(x.3+x.4)/2) else p.3;
    se = if p=ε then f(x.2,x.3) else p.4;
    sw = if p=ε then f(x.1,x.3) else p.5;
    w = if p=ε then f(x.1,(x.3+x.4)/2) else p.6;
    n = f((x.1+x.2)/2,x.4); s = f((x.1+x.2)/2,x.3); z = f((x.1+x.2)/2,(x.3+x.4)/2);
end;
agent R2 = chan tree(r2) d → chan tree(r2) d1, d2, chan r2 d3 :
    d1 = left.d; d2 = right.d; d3 = head.d;
end;
agent HB2 = chan r4i x, chan r6 p, chan r2 d → chan real t, chan r3 u, chan r6 p3, p4 :
    t = Expr{sw, nw, se, ne, s, n, w, e, z, x.1, x.2, x.3, x.4};
    u = [w,z,e]; p3 = [nw,n,ne,e,z,w]; p4 = [w,z,e,se,s,sw];
    nw = if p=ε then f(x.1,x.4) else p.1;
    n = if p=ε then f((x.1+x.2)/2,x.4) else p.2;
    ne = if p=ε then f(x.2,x.4) else p.3;
    se = if p=ε then f(x.2,x.3) else p.4;
    s = if p=ε then f((x.1+x.2)/2,x.3) else p.5;
    sw = if p=ε then f(x.1,x.3) else p.6;
    w = if d=ε then f(x.1,(x.3+x.4)/2) else d.1;
    e = if d=ε then f(x.2,(x.3+x.4)/2) else d.2;
    z = f((x.1+x.2)/2,(x.3+x.4)/2);
end;
agent T = chan tree(r3) u1, u2, chan r3 u3 → chan tree(r3) uu :
    left.uu = u1; right.uu = u2; head.uu = u3;
end;
agent IF7 = chan r4i x, chan real t, chan r6 pp1, pp2, chan tree(r2) uu1, uu2, uu3, uu4→
```

```
                                   chan real y,chan r6 p1, p2, chan tree(r2) u1, u2, u3, u4 :
        (y, p1, p2, u1, u2, u3, u4) = if x.5 = 0 then (0, ε, ε, ε, ε, ε, ε)
                     else (t, pp1, pp2, uu1, uu2, uu3, uu4) fi;
    end;
    agent IF2 = chan r4i x, chan real t, chan tree(r3) uu → chan real y, chan tree(r3) u :
        (y, u) = if x.5 = 0 then (0, ε) else (t, uu) fi;
    end;
q = A(a, ε, ε, ε)
end;
```

The agents G1, G2, G3, H1, H2, S3, IF are omitted in Q2 because they are the same as in the program Q1.

Claim 2. The program Q2 implements the specification (1).

8.3 Simplified variant of the program

In this subsection we develop another variant of the optimized program. It corresponds to the functional program presented in [Zen92] and is simplified in comparison with the program Q2 due to the following assumptions:

- we use the following equivalent representation of the function HB:

  HB = Expr1{n, s, w1, e1, z, a1, b1, a2, b2},
  where n, s, z, a1, b1, a2, b2 denote the same values as above and

  w1 = w - 1/2(nw + sw), e1 = e - 1/2(ne + se).
  The values w1 and e1 are called hierarchical surpluses in the corresponding points of the sparse grid.

- the values of the function f at the northern and southern bounds and the surplus values at the eastern and western bounds of the original rectangle are assumed to be computed beforehand and are considered as an input of the program.

Consideration in [Zen92] uses the method of the rectangle partition which is orthogonal to our's but this does not imply principial differences. In the program Q3 we try to use variables similar to those in [Zen92].

```
program Q3 = chan r4i a, chan tree(real) nn, ss, ee, ww → chan real q :
    agent A = chan r4i x, chan tree(real) n, s, e, w → chan real y :
```

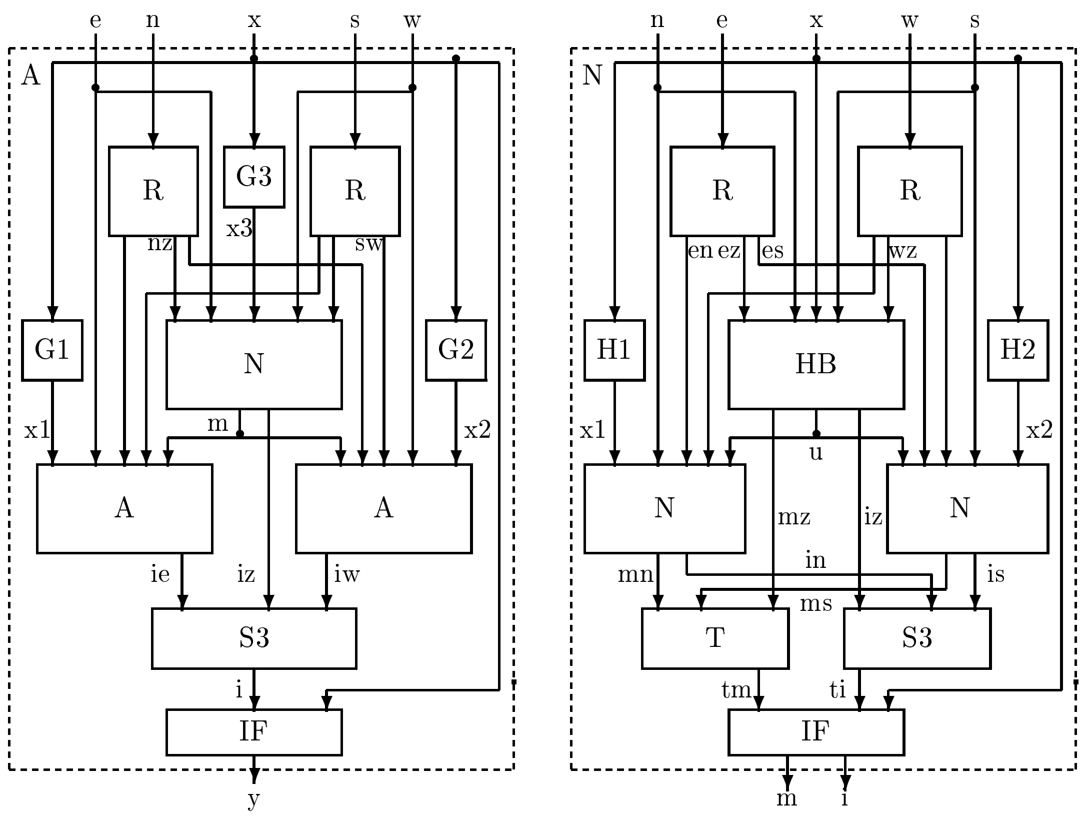

Figure 7: The program Q3

```
    y = IF(x, i); t = S3(iz, ie, iw);
    iw = A(x1, nw, sw, m, w); ie = A(x2, ne, se, e, m);
    (m, iz) = N1(x3, nz, sz, e, w);
    (nz, ne, nw) = R(n); (sz, se, sw) = R(s);
    x1 = G1(x); x2 = G2(x); x3 = G3(x);
end;
agent N = chan r4i x, chan real n, s, chan tree(real) e, w →
                              chan tree(real) m, chan real i:
    (m, i) = IF2(x, tm, ti);
    ti = S3(iz, in, is); tm = T(mz, mn, ms);
    (mn, in) = N(x1, n, u, en, wn); (ms, is) = N(x2, u, s, es, ws);
    (mz, iz, u) = HB(x, n, s, ez, wz); x1 = H1(x); x2 = H2(x);
    (ez, en, es) = R(e); (wz, wn, ws) = R(w);
end;
agent HB = chan r4i x, chan real n, s, e, w → chan real m, i, u :
        u = f((x.1+x.2)/2, (x.3+x.4)/2); m = u - (n+s)/2;
        i = Expr1{n, s, e, w, u, x.1, x.2, x.3, x.4};
    end;
    agent R = chan tree(real) t → real z, chan tree(real) l,r :
        z = head.t; l = left.t; r = right.t;
    end;
    agent T = chan real z, chan tree(real) l, r → chan tree(real) t :
        head.t = z; left.t = l; right.t = r;
    end;
    agent IF2 = chan r4i x, chan tree(real) tm, chan real ti → chan tree(real) m, chan real i :
        (m, i) = if x.5 = 0 then (ε, 0) else (tm, ti) fi;
    end;
q = A(a, n, s, e, w)
end;
```

References

[BDD$^+$92a] M. Broy, F. Dederichs, C. Dendorfer, M. Fuchs, T. F. Gritzner, and R.Weber. The design of distributed systems - an introduction to FOCUS. Technical Report SFB-Nr. 342/2/92, Techn. Univ. Muenchen, January 1992.

[BDD$^+$92b] M. Broy, F. Dederichs, C. Dendorfer, M. Fuchs, T. F. Gritzner, and R. Weber. Summary of case studies in FOCUS — a design method for di stributed systems. SFB-Nr. 342/3/92 A, Techn. Univ. Muenchen, January 1992.

[Bro89] M. Broy. Towards a design methodology for distributed systems. In M. Broy, editor, *Constructive Methods in Computer Science*, volume 55 of *NATO ASI Series F*, pages 311–364. Springer, 1989.

[CCL89] M. Chen, Y. Choo, and J. Li. Theory and pragmatics of compiling efficient parallel code. Technical Report TR-760, Yale University, 1989.

[Den85] J. Dennis. Data flow computation. In M. Broy, editor, *Control Flow and Data Flow*, volume 14 of *NATO ASI Series F: Computer and System Sciences*, pages 346–397. Springer, 1985.

[FCO90] J. Feo, D. Cann, and R. Oldehoeft. A report on the Sisal languge project. *Journal of Paral. and Distr. Comp.*, 10:349–366, 1990.

[FT89] I. Foster and S. Taylor. *Strand: New concepts in parallel programming.* Prentice-Hall, Englewood Cliffs, N.J., 1989.

[Gri90] M. Griebel. A parallelizable and vectorizable multi-level algorithm on sparse grids. Technical report, Techn. Univ. Muenchen, October 1990.

[KLGG89] Yu. Kapitonova, A. Letichevsky, S. Gorlatch, and G. Gorlatch. The use of the equations over data structures for program specification and synthesis. *Cybernetics*, (1):22–35, 1989.

[PZ81] A. Pnueli and R. Zarhi. Realizing an equational specification. *Lect.Notes Comp.Sci.*, 115:459–478, 1981.

[RW89] Th. Ruppelt and G. Wirtz. Automatic transformation of high-level specifications into parallel programs. *Parallel Computing*, 10:15–28, 1989.

[TSSP85] J. Tseng, B. Szymanski, Y. Shi, and N. Prywes. Real-time software cycle with the Model system. *IEEE Trans. Software Engin.*, 11:1136–1140, 1985.

[WA85] W. Wadge and E. Ashcroft. *Lucid, the dataflow programming language.* Academic Press, 1985.

[Zen90] Chr. Zenger. Sparse grids. Technical Report SFB-Nr. 342/18/90 A, Techn. Univ. Muenchen, October 1990.

[Zen92] Chr. Zenger. Funktionale Programmierung paralleler Algoritmen. Unpublished paper, 1992.