Induction-Oriented Formal Verification in Symmetric Interconnection Networks

Eric Gascard and Laurence Pierre

LIM - CMI / Université de Provence - 39, rue Joliot-Curie
13453 Marseille Cedex 13 - FRANCE
{Eric.Gascard,Laurence.Pierre}@lim.univ-mrs.fr

Abstract. The framework of this paper is the formal specification and proof of *applications distributed on symmetric interconnection networks*, e.g. the torus or the hypercube. The algorithms are distributed over the nodes of the networks and use well-identified communication primitives. Using the notion of *Cayley graph*, we model the networks and their communications in the inductive theorem prover *Nqthm*. Within this environment, we mechanically perform correctness verifications with a specific invariant oriented method. We illustrate our approach with the verification of two distributed algorithms implemented on the hypercube.

1 Introduction

We propose a methodology for the specification and the formal verification of distributed algorithms designed to be implemented on symmetric interconnection networks such as the ring, the torus or the hypercube [14]. In areas like signal processing or computer vision, the necessity of improving the performance of the applications is increasing inexorably. The parallelization of the algorithms in view of an implementation on specific interconnection networks[1] provides a solution to this problem [15,16]. The program is distributed over the nodes of the network, and the processes can exchange data by means of various types of communication procedures.

Our ultimate goal is to develop a specialized environment for the design and validation of such hardware/software architectures. To that aim, various aspects have to be taken into account: *(i)* the formal specification of the *hardware structures* and the description and validation of the usual *communication operations* over these architectures, *(ii)* the development of a methodology for the specification and the formal proof of *application programs* that make use of these communication functions, *(iii)* to make the approach accessible to application programmers, it is necessary to provide a way of mechanizing the transformation from their *source code to the formal model*.

[1] For instance, the ring of the HP/Convex SPP-1200, the tori of the Cray T3E and of the Intel Paragon XP, or the hypercube of the nCUBE 3.

T. Margaria and T. Melham (Eds.): CHARME 2001, LNCS 2144, pp. 418–432, 2001.

All these aspects depend on the formal proof tool that we choose. In the framework of regular structures like symmetric interconnection networks, the induction-based prover *Nqthm* [4] provides valuable assistance. Its mechanisms allow to reason at a high-level of abstraction and to deal with parameterization (here the size of the network is a parameter). We model interconnection networks in Nqthm as *Cayley graphs* [1], which give a representation of the mathematical groups defined by generators: the vertices are the elements of the group and the arcs are the actions of the generators. Cayley graphs have been extensively used to solve problems of routing [2] or information dissemination [8,3] in interconnection networks. To our knowledge, this model has never been applied to formal verification. We will show that one of the advantages of this approach is that it allows to reason independently on the structures on the one hand and on the algorithms on the other hand. Moreover, most of the specifications and verifications are performed at the processor level i.e., the statements of most theorems only involve one (universally quantified) processor, as Sects. 5 and 6 will demonstrate.

After a brief presentation of the networks and their communication primitives in Sects. 2 and 3, we describe our models and proof methodology in Sects. 4 and 5. We concentrate on the hypercube, and we make the hypothesis that the behaviour of the processes that constitute the application is synchronous (i.e., either they run on a synchronous SIMD computer, or they are designed for a MIMD architecture and include resynchronization phases). Verifying the correctness of such a distributed program is understood here as proving its equivalence with its sequential counterpart. Section 6 provides two illustrative examples, Sect. 7 recalls some related works, and we conclude in Sect. 8.

2 Cayley Graphs and Interconnection Networks

Interconnection networks can be modelled by finite graphs: the vertices represent the nodes and the edges are the communication lines. A special class of networks, called *symmetric interconnection networks*, has the property that the network viewed from any vertex looks the same (*Vertex Symmetry*). The *Cayley graph* model is proposed in [1] for designing and analyzing symmetric interconnection networks, such as the cycle, the torus and the hypercube presented here.

2.1 Cayley Graphs

Definition 1. *Let G and S be a group and a subset of this group, the Cayley digraph of G and S, denoted $Cay(G, S)$, is such that its vertices are the elements of G and its arcs are all ordered pairs $(g, g \otimes s)$ where $g \in G$, $s \in S$ and $\otimes$ is the law of the group.*

The elements of a subset S of a group G are called *generators* of G, and S is said to be a *generating set*, if every element of G can be expressed as a finite product of their powers. We say that G is generated by S.

If S is a generating set of G, then the digraph $Cay(G, S)$ is strongly connected. If S is unit free (does not contain the identity) and closed under inverses (if $s \in S$, then $s^{-1} \in S$) then $Cay(G, S)$ is a graph.

2.2 Cycle and Torus

The **cycle/ring** of length n, C_n, is the graph whose nodes are labelled by integers ranging from 0 to $n-1$ and whose edges connect i $(0 \leq i < n)$ to $(i+1)$ mod n, its degree is 2. It is the Cayley graph of the additive groupe $\mathbb{Z}_n$ generated by $\{-1, +1\}$. The actions of the generators g_1 and g_2 of C_n on a vertex s are expressed by $s \otimes g_1$ and $s \otimes g_2$, that we model by $g_1(s, n)$ and $g_2(s, n)$:

$$\begin{cases} g_1(s,n) \ = \ (s \ + \ (n-1)) \bmod n \ \text{ if } \ s \in [0, n-1] \\ g_2(s,n) \ = \ (s+1) \bmod n \ \text{ if } \ s \in [0, n-1] \end{cases}$$

The **torus** $T_{n,m}$ is the cartesian product of two cycles of lengths n and m. $T_{n,m}$ has $n \times m$ vertices, its degree is 4. It is the Cayley graph of the group $\mathbb{Z}_n \times \mathbb{Z}_m$ generated by $S = \{(0,1), (0,-1), (1,0), (-1,0)\}$. See the torus $T_{3,3}$ on Fig. 1.

2.3 Hypercube

The n-dimensional **hypercube** H_n consists of 2^n processors, its degree is n. Its nodes can be labelled by binary strings $(x_1 x_2 \ldots x_n)$; there is an edge between two nodes if their labels differ in exactly one position (the dimension). H_n is also the Cayley graph of the permutation group G generated by the n transpositions $g_i = \langle 2i-1, 2i \rangle$, $1 \leq i \leq n$ on the set $X = \{1 \ldots 2n\}$. With this view, each vertex is a permutation $(a_1, a_2, \ldots, a_{2n})$ such that $(a_{2i-1}, a_{2i}) = (2i-1, 2i)$ if $x_i = 0$ and $(a_{2i-1}, a_{2i}) = (2i, 2i-1)$ if $x_i = 1$, see Fig. 2.

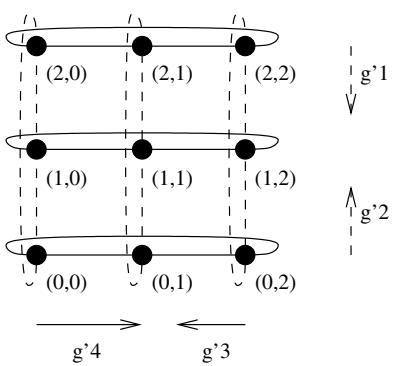

Fig. 1. Cayley graph for the torus $T_{3,3}$

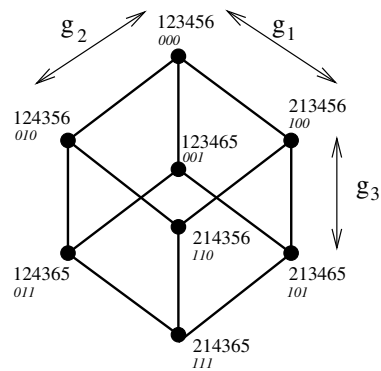

Fig. 2. Cayley graph for the hypercube H_3

3 Communication Operations

We have developed Nqthm libraries for some widespread communication primitives on these interconnection networks. Let us briefly recall them, more details can be found in [14,9].

- *Broadcast:* a node (processor) sends the same message to every other node.
- *Scatter:* a node distributes portions of a message among all the other nodes.
- *Reduction:* assume that each processor P_i, $1 \leq i \leq N$, holds a value v_i. Let $\oplus$ be a user-specified commutative, associative operation on the elements v_i. After the reduction computation, each processor P_i holds the value $\oplus_{k=1}^{N} v_k$.
- *Reduction-to-one:* the reduction-to-one operation is similar but deposits the result $\oplus_{k=1}^{N} v_k$ on a specified node.
- *Prefix Like Computation:* it is also similar to the reduction, but each processor P_i ultimately holds the value $\oplus_{k=1}^{i} v_k$.
- *Fold:* assume that each processor P_i holds a vector z_i of length l. After the fold operation, each processor P_i holds a part of length l/N of $\sum_{k=1}^{N} z_k$.
- *Expand:* conversely, after the expand operation, each processor P_i holds the vector of length $l * N$ which is the concatenation of all the vectors z_i.

4 Modelling Methodology

Our purpose is to model in the Boyer-Moore logic the networks presented in Sect. 2 and their ad hoc realizations of the communication operations of Sect. 3. Every concept is given a purely functional representation.

4.1 Modelling the Networks

Every interconnection network is characterized by generators. The function that computes s', the label of the node that can be reached by applying the generator g_i to the node s ($s' = s \otimes g_i$) in a graph of size n will be referred to as g_i, i.e. $s' = g_i(s, n)$. We define a function *Cayley* for each graph, it returns its set of vertices. In the case where the number of generators is fixed whatever the size is (e.g. the cycle or the torus) the Cayley graph can be defined as follows: starting from $I(n)$, the identity element of σ_n (the group of all permutations over n elements), we iteratively compute the vertices. At each step, new vertices are computed, by application of the generators g_i to the vertices obtained in the previous iterations. The construction stops when no new vertex can be obtained. The necessary number of steps is equal to the diameter of the graph. In the case where the number of generators is a function of the size of the Cayley graph i.e., the graph is hierarchical (e.g. the hypercube), an alternative approach described below can be taken.

Definition 2. *The graph $G(n)$, of dimension n, is hierarchical if there exists a non null function $\mathcal{K}$ such that for $n > 1$, $G(n)$ is decomposable into $\mathcal{K}(n)$ disjoint hierarchical subgraphs that are isomorphic to $G(n-1)$.*

We call $Iso(i, G(n))$, $1 \leq i \leq \mathcal{K}(n)$, the i-th subgraph isomorphic to $G(n)$. $G(n)$ can be associated with $Cayley(n, n)$, where *Cayley* is defined as follows (k is the level of decomposition):

$\underline{\textbf{funct}}\ Cayley(k, n)\ \equiv$
$\underline{\textbf{if}}\ k = 0\ \underline{\textbf{then}}\ \{I(n)\}\ \underline{\textbf{else}}\ \bigcup_{1 \leq i \leq \mathcal{K}(k)} Iso(i, Cayley(k-1, n))\ \underline{\textbf{fi}}$

This methodology has been used to encode the interconnection networks in Nqthm. For the hypercube, the function above is instanciated with $\mathcal{K}$ as the constant function 2. We have also verified basic properties for each graph, such as its finiteness and its vertex symmetry. Table 1 gives the corresponding numbers of Nqthm events (an *event* is a function definition or a theorem).

Table 1. Number of events to model Cayley graphs in Nqthm

	# definitions	# intermediate lemmas	# theorems
Cycle	14	60	16
Torus	26	108	20
Hypercube	14	32	3

4.2 Modelling the Neighbourhood of Communicating Processors

Our proof methodology makes use of *invariants* that call on the notion of *neighbourhood* of a vertex. $Neighb(s,t,n)$ represents the set of vertices that have communicated, directly or not, with the vertex s at step t of the communication algorithm running on a network of size n. Depending on the communication operation, one or several direct neighbour(s), reachable by one or several generators, are added to this set at each step; this is characterized by the set $C_t \equiv \{\, i \mid g_i(s,n)$ communicates with s at step $t \,\}$.

$$\begin{array}{l}\underline{\textbf{funct}}\ Neighb(s,t,n)\ \equiv\\ \underline{\textbf{if}}\ t=0\ \underline{\textbf{then}}\ \{s\}\\ \qquad\qquad \underline{\textbf{else}}\ Neighb(s,t-1,n) \cup \bigcup_{i\in C_t} Neighb(g_i(s,n),t-1,n)\\ \underline{\textbf{fi}}\end{array}$$

Example. Neighbourhood for the broadcast in the hypercube. At every step t, each processor communicates with its direct neighbour in the t^{th} dimension, see Fig. 3 where each number represents both the dimension and the step number.

$$\begin{array}{l}\underline{\textbf{funct}}\ Neighb_{bh}(s,t,n)\ \equiv\\ \underline{\textbf{if}}\ t=0\ \underline{\textbf{then}}\ \{s\}\\ \qquad\qquad \underline{\textbf{else}}\ Neighb_{bh}(s,t-1,n)\\ \qquad\qquad\qquad \cup\ Neighb_{bh}(g_t(s,n),t-1,n)\\ \underline{\textbf{fi}}\end{array}$$

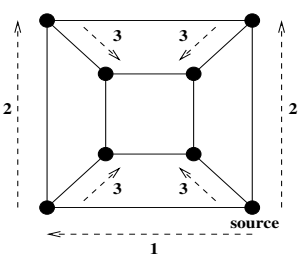

Fig. 3. Broadcast on H_3

4.3 Modelling the Communications

It remains to formalize, for every communication operation and every architecture, the code executed by each processor. For each operation, we define a function that takes the following arguments:

- s: the processor where the algorithm is running;
- k: the number of iterations;
- ini: initial state of the system (associative list of pairs of the form (s, val_{init}) where s is a vertex and val_{init} is the initial state of its local register(s)). It is associated with a function *Take_value* such that $Take\_value(s, ini) = val_{init}$;
- n: the size of the network.

The function returns the value(s) stored in the register(s) of the processor after k iterations. For instance, a function $Broadcast_h(s, k, ini, n)$ is defined to model the broadcast operation on a n-dimensional hypercube, see its Nqthm definition in the next section.

4.4 Link with an MPI-Based Implementation

In this section we consider the link between our formal model and a source code written in a usual programming language. This aspect is fundamental w.r.t. a possible integration of our approach in a development environment. The overall method described in this paper is adapted to the validation of applications designed to run on distributed memory SIMD or MIMD target architectures, provided that synchronization takes place between the elementary point-to-point send/receive operations. MPI (Message Passing Interface) [20] is a standard for the realization of message passing procedures; MPICH is one of its implementations that allows to develop applications on a network of workstations. In view of designing a methodology of transformation from a C program, using MPI point-to-point primitives, to our formalism, we study the correspondance between these two representations in the case of the collective operations presented in Sect. 3.

Here is our principle for implementing every procedure: since each processor cannot know in which iteration it will receive (and then send) the message which is broadcasted, scattered and the like, it executes "send" and "receive" operations at every step. It receives both a message and a "flag" that indicates if this message is relevant. In that case, it updates its register(s) and sets its own flag to true. The C version of the *broadcast* function for H_n is given below.

```
int broadcast_h (char *s, int n, void *buffer, int count,
                 MPI_Datatype datatype, int *flag, void *mybuffer,
                 int *myflag, MPI_Comm comm)
{ char *neighb; /* neighbour in the k-th dimension */
  int k;  MPI_Status st;
  MPI_Aint l; MPI_Type_extent(datatype,&l);
  for (k=0; k<n; k++)
  { /* transfer of data with the neighbour: */
    neighb = gen_hyper(k, s, n); /* permutation of the neighbour */
    MPI_Send(mybuffer,count,datatype,PermutationToInt(neighb),k,comm);
    MPI_Send(myflag,1,MPI_INT,PermutationToInt(neighb),k,comm);
    MPI_Barrier(comm);    /* synchronization */
```

```
    MPI_Recv(buffer,count,datatype,PermutationToInt(neighb),k,comm,&st);
    MPI_Recv(flag,1,MPI_INT,PermutationToInt(neighb),k,comm,&st);
    if (Hyper_sendp (*myflag)) /* if my flag is 1, data are unchanged */
      { }
    else if (Hyper_sendp (*flag))
         /* if the neighbour's flag is 1, I update my data */
           {  memcpy(mybuffer, buffer, count*l);
              *myflag = 1;
           }
         /* otherwise, data are unchanged */
  }
  return 1;
}

int Hyper_sendp (int flag) { return (flag==1); }
```

The Nqthm function `broadcast_h` (see below) is its faithful translation; it is defined as described in the previous section. Its definition is recursive, its parameter `k` corresponds to the iteration variable in the C function. It includes standard hypotheses on the types and constraints related to the parameters, systematically introduced in each function. Its second `let` gives the values of the variables that characterize the data to be sent (though the "send" operation is not explicit in our functional representation). The third `let` corresponds to the variables associated with the data that are received from the neighbour(s). In both cases, the Lisp expressions can be generated in a systematic way from the C source, knowing the names of the Nqthm *accessor* functions (`buf-rcv`, `buf-snd`, ...). A test is included to stop recursion; its condition corresponds to the initialisation of the iteration variable in the C program, and the returned value is given by the expression that computes the initial value of the set of buffers for processor `s` (in `ini`). In the rest of the function body, there are as many nested "if" statements as in the C function:

- in the case(s) where data are updated in the C function, the Lisp function returns the new status of the set of buffers for processor `s`, using the expression `(cbuffer buffer (message t buffer))` which means that the new value of `mybuffer`, resp. `myflag`, is `buffer`, resp. 1 (value `t`). In the broadcast operation, the message to be sent is exactly the one that has been received.
- in the case(s) where data are unchanged in the C function, the value of the set of buffers remains the same and the Lisp function recurses with `k` decreased by 1. In the broadcast example, the corresponding expression is `(broadcast_h s (sub1 k) ini n)`.

```
(defn broadcast_h (s k ini n)
 (if (and (not (zerop n)) (numberp k) (leq k n)
          ...)  ; + other hypotheses
   (let ((neighb (gen_hyper k s n)))  ; neighbour
     (let (; data to be sent:
           (mybuffer (buf-rcv (broadcast_h s (sub1 k) ini n)))
```

```
          (myflag (get-flag (buf-snd (broadcast_h s (sub1 k) ini n)))))
      (let (; data received:
            (buffer (get-msg (buf-snd
                              (broadcast_h neighb (sub1 k) ini n))))
            (flag (get-flag (buf-snd
                              (broadcast_h neighb (sub1 k) ini n)))))
        (if (zerop k)   ; to stop recursion
            (take_value s ini)
            (if (hyper_sendp myflag) ; data are unchanged
                (broadcast_h s (sub1 k) ini n)
                (if (hyper_sendp flag)
                    ; the new buffer contains ``buffer'' and ``true'':
                    (cbuffer buffer (message t buffer))
                    ; otherwise data are unchanged
                    (broadcast_h s (sub1 k) ini n)))))))
  (empty)))

(defn hyper_sendp (flag) (equal flag t))
```

5 Proof Methodology

5.1 Description of the Method

Our proof method is the same for every communication primitive and every network. First we exhibit and prove an invariant $\mathcal{I}$ and then, using a proof obligation $\mathcal{O}$ that is related to the function $Neighb$, we complete the proof of the main theorem $\mathcal{T}$. The invariant as well as the characteristics of $Neighb$ depend on the problem, but we will show that $\mathcal{I}$ can easily be deduced from $\mathcal{T}$. The form of $\mathcal{T}$, $\mathcal{I}$ and $\mathcal{O}$ clearly demonstrates that the verifications are performed at the processor level; they involve *one* (universally quantified) processor s. In the following, we assume that the graph is built after t_c iterations, $Comm$ denotes the communication function which needs t_f iterations to complete, and S represents the specification function. The form of $\mathcal{T}$ slightly differs according to whether the operation involves a source processor or not. In the case where there is no specific source, the equality $\mathcal{T}$ to be proven is of the form:

$$\forall s \in Cayley(t_c, n).\, Comm(s, t_f, ini, n) = S(s, Cayley(t_c, n), ini, n)$$

The statement of $\mathcal{I}$ generalizes the one of $\mathcal{T}$ to k iterations, considering the *neighbourhood* of s. After k iterations, s holds the value given by the specification function S applied to the values of the processors in its neighbourhood:

$$\forall s \in Cayley(t_c, n), \forall k \leq t_f.\ Comm(s, k, ini, n) = \ S(s, Neighb(s, k, n), ini, n)$$

The proof obligation $\mathcal{O}$ necessary to get $\mathcal{T}$ states that, at the end of the operation, every processor has taken part in the communication:

$$\forall s \in Cayley(t_c, n).\, Neighb(s, t_f, n) = Cayley(t_c, n)$$

When the operation involves a source (e.g. broadcast), the idea is similar but the theorems make use of the source, and they assume its uniqueness.

Table 2 gives the numbers of Nqthm events for the communication operations that we have validated using this methodology (the first column is for the specification of the "neighbourhoods", "-" means that the proofs are under development).

Table 2. Communication operations in Nqthm (numbers of events)

	Neighb.	Broadcast	Reduction	Scatter	Gather	Prefix	Fold	Expand
Cycle	74	53	11	-	-	-	-	-
Torus	108	31	12	-	-	-	-	-
Hypercube	75	21	3	12	19	63	103	51

5.2 Example: Broadcast on the Hypercube

To validate the broadcast operation on H_n we prove that, after n steps, every processor has received the message sent by the source.

$\forall s \in Cayley_h(n,n).\ Uniquenessp(Source(ini)) \implies$
$Broadcast_h(s,n,ini,n) = Take\_Value(Source(ini), ini)$

The invariant $\mathcal{I}$ generalizes this statement:

$\forall s \in Cayley_h(n,n), \forall k \leq n.\ Uniquenessp(Source(ini)) \implies$
$Broadcast_h(s,k,ini,n) =$
if $s \in Neighb_{bh}(Source(ini),k,n)$ **then** $Take\_Value(Source(ini),ini)$
else $Take\_Value(s,ini)$

The lemma $\mathcal{O}$ states that, after n steps, every processor is in the neighbourhood of the source.

$\forall s \in Cayley_h(n,n).\ s \in Neighb_{bh}(Source(ini),n,n)$

6 Applicative Examples

The libraries we have developed can be used for the verification of application programs (we verify that they are equivalent to their sequential counterparts).

6.1 Computational Geometry Algorithm

This algorithm [19] solves a problem of computational geometry: given a query point z, determine whether it lies in a region R. Here, z is a planar point and R is a polygon. For a N-polygon, the program can be implemented on a hypercube with $N = 2^n$ processors. Initially, a source processor s holds the query point z and the set E of the N edges e_i that defines the polygon. The algorithm is:

1. the source s *scatters* the edges over the network, one edge per processor
2. it *broadcasts* the query point to all other processors
3. each processor P_i calls on a function *Intersectp* that returns $l_i = 1$ (otherwise 0) if its edge intersects the horizontal line containing z to the left of z
4. every processor computes $\sum_{i=1}^{N} l_i$; it is a *reduction* procedure with the addition as operator. If this sum is odd then z is internal to the polygon.

Each processor has two buffers, their values are enclosed in brackets in the function definitions below and are accessible by the functions acc_1 and acc_2. We model each step of the algorithm by a function $Step_i$. The expressions $Build\_ini_i(\ldots)$ give the global state of the network after the step number i.

1. The source processor s sends to each processor P_i the edge e_i:

   $Step1(P_i, s, E, z, n) \equiv$
   $[\ Scatter_h(P_i, n, Build\_ini_0(s, E), n)\ ;\ \underline{\textbf{if}}\ P_i = s\ \underline{\textbf{then}}\ z\ \underline{\textbf{else}}\ \perp]$

   Correctness lemma: each P_i has received the edge e_i.

   $\forall P_i, s \in Cayley(n, n).\ Step1(P_i, s, E, z, n) = [e_i\ ;\ \underline{\textbf{if}}\ P_i = s\ \underline{\textbf{then}}\ z\ \underline{\textbf{else}}\ \perp]$

2. The processor s broadcasts z to every other processor:

   $Step2(P_i, s, E, z, n) \equiv$
   $[\ acc_1(Step1(P_i, s, E, z, n))\ ;$ *contents of the 1st buffer unchanged*
   $Broadcast_h(P_i, n, Build\_ini_1(\ldots), n)\ ]$

   Correctness lemma: each P_i has received z.

   $\forall P_i, s \in Cayley(n, n).\ Step2(P_i, s, E, z, n) = [\ e_i\ ;\ z\ ]$

3. Each processor P_i computes l_i:

   $Step3(P_i, s, E, z, n) \equiv$
   $[\ Intersectp(acc_1(Step2(P_i, s, E, z, n)),\ acc_2(Step2(P_i, s, E, z, n)))\ ;$
   $acc_2(Step2(P_i, s, E, z, n)))\ ]\ ;$ *contents of the 2nd buffer unchanged*

   Correctness lemma: each P_i has correctly computed its l_i.

   $\forall P_i, s \in Cayley(n, n).\ Step3(P_i, s, E, z, n) = [\ Intersectp(e_i, z)\ ;\ z\ ]$

4. Each processor computes $\sum_{k=1}^{2^n} l_k$:

   $Step4(P_i, s, E, z, n) \equiv$
   $[\ Reduction\_Sum_h(P_i, n, Build\_ini_3(\ldots), n)\ ;$
   $acc_2(Step3(P_i, s, E, z, n)))\ ]\ ;$ *contents of the 2nd buffer unchanged*

   Correctness of the complete algorithm: each processor knows if z is internal or external to the polygon.

   $\forall P_i, s \in Cayley(n, n).\ Step4(P_i, s, E, z, n) = [\ \sum_{k=1}^{2^n} Intersectp(e_k, z)\ ;\ z\ ]$

6.2 Matrix-Vector Product

Let us consider the algorithm proposed in [9] to perform a matrix-vector multiplication on the hypercube. Consider the product $y = Ax$ where A is an $n \times n$ matrix and x and y are vectors of length n. The algorithm computes y on H_d (there are $p = 2^d$ processors, n is evenly divisible by p, and d is even i.e., $\sqrt{p} = 2^{\frac{d}{2}}$ is a natural number).

Decomposition and Assignment of Data. The matrix A is decomposed into square blocks of size $(n/\sqrt{p}) \times (n/\sqrt{p})$. Each block is denoted $\mathcal{A}[\alpha, \beta]$, α, β running from 0 to $\sqrt{p}-1$, see Fig. 4. The input vector x and product vector y are also divided into $\sqrt{p}$ pieces. The subvector number β of x, resp. the subvector number α of y, will be denoted $\mathcal{X}[\beta]$, resp. $\mathcal{Y}[\alpha]$. Each pair of subscripts (α, β) can be associated with one of the p processors of H_d, which will be in charge of computing its contribution to $\mathcal{Y}[\alpha]$ in terms of $\mathcal{A}[\alpha, \beta]$ and $\mathcal{X}[\beta]$. This processor will be referred to as $P_{\alpha\beta}$; its registers initially holds the data $\mathcal{A}[\alpha, \beta]$ and $\mathcal{X}[\beta]$.

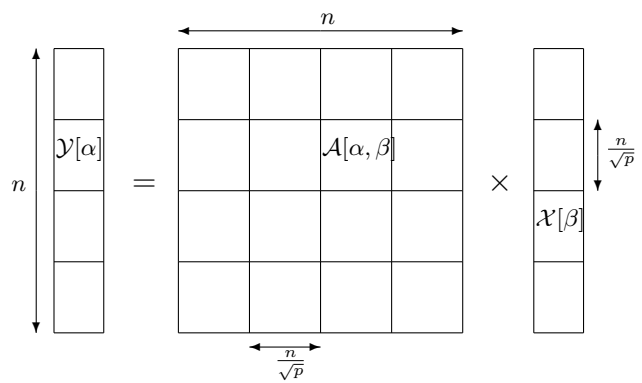

Fig. 4. Distribution of data

Simplified Matrix–Vector Product. The method is the following (see Fig. 5 for the algorithms of the *fold* and *expand* operations):

- $P_{\alpha\beta}$ computes its contribution to $\mathcal{Y}[\alpha]$. This is a vector of length $n/\sqrt{p}$ which we denote by $\mathcal{Z}[\alpha, \beta]$; thus $\mathcal{Z}[\alpha, \beta] = \mathcal{A}[\alpha, \beta] \times \mathcal{X}[\beta]$ and $\mathcal{Y}[\alpha] = \Sigma_\beta \mathcal{Z}[\alpha, \beta]$

- The second step uses the *fold* operation to sum vectors $\mathcal{Z}[\alpha, \beta]$. This operation is executed between a group of $\sqrt{p}$ processors; it requires $log_2(\sqrt{p}) = \frac{d}{2}$ steps, halving the length of the vectors involved at each step. Within each step, a processor first divides its vector z into two equal size subvectors, z_1 and z_2 (notation $(z_1|z_2)$). One of these subvectors is sent to another processor P, and a subvector w is received from P. It is summed element-by-element with the retained subvector. As a result, each processor $P_{\alpha\beta}$ has a unique, n/p-length portion of the fully summed vector denoted $\mathcal{Y}[\alpha]_{[\beta]}$.

– The last step uses the *expand* operation. Each processor in the row has a subvector $z = \mathcal{Y}[\alpha]_{[\beta]}$. At each step, it sends z to another processor P and receives w from P. These two subvectors are concatenated in the correct order, as indicated by the notation '|', to form the updated value of z. This operation requires $log_2(\sqrt{p})$ steps. This primitive combines the subvectors $\mathcal{Y}[\alpha]_{[\beta]}$ and places the complete vector $\mathcal{Y}[\alpha]$ in each processor of row α.

Fold operation for $P_{\alpha\beta}$	Expand operation for $P_{\alpha\beta}$
$z := \mathcal{Z}[\alpha, \beta]$	$z := \mathcal{Y}[\alpha]_{[\beta]}$
for $i := 0$ **to** $log_2(\sqrt{p}) - 1$	**for** $i := log_2(\sqrt{p}) - 1$ **to** 0
$(z_1 \mid z_2) = z$	$P := P_{\alpha\beta}$ with i^{th} bit of β flipped
$P := P_{\alpha\beta}$ with i^{th} bit of β flipped	**Send** z to processor P
if bit i of β is 1 **then**	**Receive** w from processor P
Send z_1 to processor P	**if** bit i of β is 1 **then**
Receive w from processor P	$z := w \mid z$
$z := z_2 + w$	**else**
else	$z := z \mid w$
Send z_2 to processor P	$\mathcal{Y}[\alpha] := z$
Receive w from processor P	
$z := z_1 + w$	
$\mathcal{Y}[\alpha]_{[\beta]} := z$	

Fig. 5. Communication primitives for processor $P_{\alpha\beta}$

As in the previous example, we model each step of the algorithm by a function $Step_i$, which is associated with an intermediate correctness lemma.

1. Each processor $P_{\alpha\beta}$ of a d-dimensional hypercube computes locally a matrix-vector product. The expression $acc_1(Take\_value(P_{\alpha\beta}, Build\_ini_0(\ldots)))$ returns $\mathcal{A}[\alpha, \beta]$ and $acc_2(Take\_value(P_{\alpha\beta}, Build\_ini_0(\ldots)))$ returns $\mathcal{X}[\beta]$.

$$Step1(P_{\alpha\beta}, A, X, d) \equiv Matrix\_Vector\_Product(acc_1(Take\_value(P_{\alpha\beta}, Build\_ini_0(\ldots))) \; acc_2(Take\_value(P_{\alpha\beta}, Build\_ini_0(\ldots))))$$

2. $P_{\alpha\beta}$ computes $\mathcal{Y}[\alpha]_{[\beta]}$ by applying the fold operation. The function $Build\_ini_1$ returns the global state of the network after the first step.

$$Step2(P_{\alpha\beta}, A, X, d) \equiv Fold(P_{\alpha\beta}, \tfrac{d}{2}, Build\_ini_1(\ldots), d)$$

Correctness lemma: each $P_{\alpha\beta}$ has received $\mathcal{Y}[\alpha]_{[\beta]}$.

$$\forall P_{\alpha\beta} \in Cayley(d, d).\, Step2(P_{\alpha\beta}, A, X, d) = (\textstyle\sum_{e \in Neighb(P_{\alpha\beta}, \frac{d}{2}, d)} Matrix\_Vector\_Product(\mathcal{A}[\alpha_e, \beta_e], \mathcal{X}[\beta_e]))_{[\beta]}$$

3. The processor $P_{\alpha\beta}$ computes $\mathcal{Y}[\alpha]$ by applying the expand operation. The function $Build\_ini_2$ returns the global state after the second step.

$$Step3(P_{\alpha\beta}, A, X, d) \equiv Expand(P_{\alpha\beta}, \tfrac{d}{2}, Build\_ini_2(\ldots), d)$$

Correctness lemma: Processor $P_{\alpha\beta}$ has computed $\mathcal{Y}[\alpha]$.

$$\forall P_{\alpha\beta} \in Cayley(d,d).\ Step3(P_{\alpha\beta}, A, X, d) = \sum_{e \in Neighb(P_{\alpha\beta}, \frac{d}{2}, d)} Matrix\_Vector\_Product(\mathcal{A}[\alpha_e, \beta_e], \mathcal{X}[\beta_e])$$

Correctness of the complete algorithm:

$$\forall P_{\alpha\beta} \in Cayley(d,d).\ Step3(P_{\alpha\beta}, A, X, d) = \sum_{j=0}^{\sqrt{p}-1} Matrix\_Vector\_Product(\mathcal{A}[\alpha, j], \mathcal{X}[j])$$

7 Related Works

The validation of applications involving distributed processes has been widely studied. Many approaches are related to the verification of communication protocols and make use of *model checking* techniques; some others put the emphasis on the correct *parallelization* of sequential code, like [5]. Here we focus only on works which are close to ours i.e., that are concerned with the verification of the equivalence between a parallel algorithm and its sequential counterpart, by means of *theorem proving* techniques. They can be classified into two main categories.

In the first one, the algorithm is modelled as a function that takes as argument the set of data distributed in the system (its *state*) and returns the state obtained after one computation step. The proof consists in showing that the state after a number N of iterations satisfies the *specification*. The function and the data structures give an explicit view of the network. In [7], we prove the correctness of a parallel algorithm for finding the maximum of set of values on a $n \times n$ 4-neighbour torus, using Nqthm. The method uses two recursive functions to model the network; the main data structure is a list of list of natural numbers. Point-to-point communications are expressed by updating the elements of this list according to the values of their neighbours. In [11], RRL is used in conjunction with *powerlists* data structures [17]. The authors verify the *Batcher's Merge Sorting Network* and prove correct the mapping of multi-dimensional arrays expressed as powerlists into hypercube networks. The verification of the FFT with ACL2 is described in [6]; the underlying representation is also based on powerlists. This notation is well suited to the specification and verification of certain types of parallel algorithms running on regular structures. Kornerup discusses in [13] the modelling of hypercube algorithms with this notation.

The second approach makes explicit the *send* and *receive* point-to-point primitives and the network structure. The algorithm is expressed by a function which describes the updating of the *global state* of the system, which consists of the private states of the processes together with the bag of messages that are in transit (sent but not yet accepted by the destination process). This model allows the specification of asynchronous communications, which could introduce nondeterminism. To model nondeterminacy, a free variable *oracle* gives a symbolic view of the list of messages that are in transit. Hesselink explains this modelling methodology in [10]. The function that expresses the distributed algorithm takes

as parameters the global state s, the oracle *ora*, the iteration number n and the description of the network (a graph) g. The proof methodology is based on invariants describing properties of the global state, and proofs are performed with Nqthm. Arbitrary interconnection networks can be considered, the only hypothesis is that the corresponding graph is connected.

There are common aspects between these approaches and ours. Like in the first one, we consider synchronized algorithms and the point-to-point communications are implicit in the formal model. A similarity with both approaches is that we use an explicit representation of the interconnection networks. However, a major advantage of modelling these structures by means of Cayley graphs is that our proof methodology works at the processor level instead of having to reason on the whole network. The graph is not encoded as a static structure, but the processor neighbours are dynamically computed using the generator functions. In many cases, the correspondance between this processor view and the global system view is straightforward. In a near future, we plan to consider more complex applications and to improve our method with techniques like the one described in [18] to map a "uniprocessor" view to a "multiprocessor" one.

8 Conclusion

We have proposed a new approach for the validation of parallel algorithms running on symmetric interconnection networks. The proof methodology exploits their topological properties. We have built reusable libraries for reasoning about widespread networks and their collective communication primitives. Two examples have illustrated the usefulness of these libraries to perform correctness verifications for distributed programs; other applications are being developed. Table 3 gives the corresponding numbers of Nqthm events. In each case, about 70 % of events have simply been imported from the libraries (without reproving them). A part of the remaining 30 % could be generated automatically, from the contents of the source code and from the specification. However, human intervention is needed for the other ones. Since there is a common basis in the underlying reasoning related to most of them, we plan to design a development environment with an interactive user-interface where the programmer could input his source code and the specification, ask for the generation of the corresponding events by means of a mechanized translator, and then guide the proof tool when needed. To that goal, we are currently re-implementing our libraries in the up-to-date prover Acl2 [12].

Table 3. Numbers of Nqthm events for the verification of the examples

	# Events (def. + lemmas)	# Events reused from our libraries	# Events specific to the algorithm
Computational Geometry	142	111 (78 %)	31 (22 %)
Matrix-vector product	417	286 (68 %)	131 (32 %)

References

[1] S. B. Akers and B. Krishnamurthy. A Group Theoretic Model for Symmetric Interconnection Networks. *IEEE Transactions on Computers*, 38(4), 1989.

[2] B.W. Arden and K.W. Tang. Representations and Routing of Cayley Graphs. *IEEE Transactions on Communications*, 39(11), November 1991.

[3] J-C. Bermond, T. Kodate, and S. Perennes. Gossiping in Cayley Graphs by Packets. In *Proceedings of Franco-Japanese conference Brest July 95*, volume 1120 of *Lectures Notes in Computer Science*. Springer Verlag, 1996.

[4] R. Boyer and J Moore. *A Computational Logic Hand-book*. Perspectives in Computing, Vol. 23. Academic Press, Inc., 1988.

[5] R. Couturier and D. Méry. An experiment in parallelizing an application using formal methods. In *CAV'98*. LNCS 1427, Springer Verlag, 1998.

[6] R. A. Gamboa. Mechanically Verifying the Correctness of the Fast Fourier Transform in ACL2. In *Third International Workshop on Formal Methods for Parallel Programming: Theory and Applications (FMPPTA)*, 1998.

[7] E. Gascard and L. Pierre. Two Approaches to the Formal Proof of Replicated Hardware Systems using the Boyer-Moore Theorem Prover. In *Proc. International Workshop on First Order Theorem Provers (FTP'97)*, October 1997.

[8] C. GowriSankaran. Broadcasting on Recursively Decomposable Cayley Graphs. *Discrete Applied Mathematics*, 53:171–182, 1994.

[9] B. Hendrickson, R. Leland, and S. Plimpton. An Efficient Parallel Algorithm for Matrix–Vector Multiplication. *Int. J. High Speed Computing*, 7(1):73–88, 1995.

[10] W. H. Hesselink. A mechanical proof of Segall's PIF algorithm. *Formal Aspects of Computing*, 9:208–226, 1997.

[11] D. Kapur and M. Subramaniam. Automated Reasoning about Parallel Algorithms using Powerlists. In *Proc. Algebraic Methodology and Software Technology, AMAST'95*, volume 936 of *LNCS*, pages 416–430. Springer-Verlag, 1995.

[12] M. Kaufmann and J S. Moore. An Industrial Strength Theorem prover for a Logic Based on Common Lisp. *IEEE Trans. on Software Engineering*, 23(4), April 1997.

[13] J. Kornerup. Mapping Powerlists onto Hypercubes. Technical Report TR-94-04, Department of Computer Sciences, University of Texas at Austin, 1994.

[14] V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Intoduction to Parallel Computing: Design and Analysis of Algorithms*. Benjamin Cummings Pub., 1994.

[15] T.M. Kurç, C. Aykanet, and B. Özguç. A parallel scaled conjugate-gradient algorithm for the solution phase of gathering radiosity on hypercubes. *The Visual Computer*, 13(1):1–19, 1997.

[16] Y. Lee, S. Horng, T. Kao, and Y. Chen. Parallel computation of the Euclidean distance transform on the mesh of trees and the hypercube computer. *Computer Vision and Image Understanding*, 68(1):109–119, October 1997.

[17] J. Misra. Powerlist: A structure for parallel recursion. *ACM Transactions on Programming Languages and Systems*, 16(6):1737–1767, November 1994.

[18] J S. Moore. A mechanically checked proof of a multiprocessor result via a uniprocessor view. *Formal Methods in System Design*, 14(2):213–228, 1999.

[19] K. Qiu and S.G. Akl. Novel Data Communication Algorithms on Hypercubes and Related Interconnection Networks and Their Applications in Computational Geometry. Technical Report 97-415, Department of Computing and Information Science, Queen's University, Kingston, Ontario, December 1997.

[20] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI: The Complete Reference*. The MIT Press, 1996.