$\mathcal{N}$-Graphs: A Topology for Parallel Divide-and-Conquer on Transputer Networks

Sergei Gorlatch and Christian Lengauer

University of Passau, D–94030 Passau, Germany. E-mail: gorlatch@fmi.uni-passau.de

Abstract

A parallel implementation of a divide-and-conquer template (skeleton) is derived systematically from its functional specification. The implementation makes use of a new processor topology for divide-and-conquer, called $\mathcal{N}$*-graph*, which suits transputer networks well: there are not more than 4 links per processor, overlapping of computations and communication within a processor is exploited, the processor network is of an arbitrary fixed size, the load is balanced and all communications between processors are local. A parallel mergesort algorithm is used to illustrate the derivation process and the distributed parallel target program. Experimental results on a 64-node transputer network are reported.

1 Introduction

The well-known technique of divide-and-conquer has been applied to a wide variety of problems. It is used when a problem solution can be defined recursively as a function of a collection of smaller instances of the same problem, generated from the description of the original instance. Recursion terminates if the particular instance (the base case) is indivisible; it is solved by some other non-recursive method. The simplest and most common divide-and-conquer algorithms are *binary*: they have a dividing factor of 2 and a binary tree of processors as ideal topology.

Divide-and-conquer represents one of the most widely used templates of parallelism. Cole coined the term *skeleton* for a template [1]. The idea of the skeleton-based approach to parallel program development is to determine typical skeletons, find their parallel implementation and then use them as building blocks for programming.

Most attention has been paid to the problems of expressing skeletons using higher-order functions as by the group of Darlington [2] or specific language constructs like Divacon by Mou [3] and their formal transformation to a form which exposes their parallelism. There has been work on an algebraic model for divide-and-conquer by Mou and Hudak [4] and on transforming the non-linear divide-and-conquer recursion into tail recursion and then pipelining it by Harrison et al. [5, 6]. Implementations of divide-and-conquer on distributed memory multiprocessors include specialized architectures like ZAPP [7] and explicit annotations for parallelism and communication as by Kelly et al. [8]. The skeletal approach and formal transformations have also been used by Pepper and others for deriving parallel networks from functional specifications [9].

It is unreasonable to assume that a fixed-size machine will be large enough to accommodate any number of problem instances that might arise. The problem of mapping divide-and-conquer algorithms efficiently onto fixed-size multiprocessor networks with a constant-degree topology has been studied for years. There have been attempts to embed binary trees into 2D-meshes [1], 3D-meshes [10] and into binary trees of a smaller size [11]. In every case, either load balance is sacrificed or the embedding has a non-unit dilation, i.e. neighbour nodes of the virtual binary tree are not always neighbours in the physical network and, thus, the communication is not local.

In this paper, we address the problem of implementing the divide-and-conquer skeleton on transputer networks, i.e. we deal with processors that have 4 links and allow overlapping of computations with communication. Our approach is to introduce a new constant-degree topology, called $\mathcal{N}$*-graph*, and to derive a parallel implementation of divide-and-conquer on this topology with the desired properties of *fixed size*, *balanced load* and *local communication*.

Since our parallel implementation for the divide-and-conquer skeleton is intended to be a generic solution for a class of algorithms, we want it to be *provably correct*. To achieve that, we start our derivation with a generic functional specification and proceed by semantics-preserving transformations in a formal framework which guarantees the correctness of the implementation. Due to lack of space, we only sketch the derivation process.

The $\mathcal{N}$-graph-implementation can be viewed as a refinement of the general parallel divide-and-conquer schema that has been developed in [12] and whose performance on a transputer network has been studied in [13]. We illustrate it, with a parallel mergesort algorithm as an example, and report some experimental results of its implementation on a 64-nodes transputer network.

In Section 2, we introduce a generic specification of divide-and-conquer and the illustrating example and discuss the difficulties of usual methods of their implementation. In Section 3, the $\mathcal{N}$-graph topology and its properties are introduced. Section 4 outlines the formal derivation of a parallel implementation of a divide-and-conquer skeleton on $\mathcal{N}$-graphs. Section 5 is on generating an imperative target program. In Section 6, we discuss a possible generalization of the $\mathcal{N}$-graph topology for divide-and-conquer with an arbitrary dividing factor. In Section 7, experiments on a 64-transputer network and their results are described. Section 8 summarizes and discusses the results.

2 Generic Specification and Virtual Topology

As already mentioned, we restrict our consideration to the binary divide-and-conquer which in fact is the most widely used version of this technique. The binary divide-and-conquer can be specified as a *recursive function* f:

$$f(x) \;=\; p(x) \;\rightarrow\; b(x)\,;\;\; E\;(\,f(\varphi_1(x))\,,\,f(\varphi_2(x))\,) \tag{1}$$

Here, the conditional $p \rightarrow b; E$ is from Backus' FP [14]. It is equal to the usual conditional expression: **if** p **then** b **else** E, where p is called the *basic predicate* and b the *basic function*. We assume that f, b, p and φ_i are defined on the same type τ whose elements are called *domain parameters*. Expression E depends on the results of two recursive calls of f.

As a sample specification, we consider a divide-and-conquer algorithm for mergesort which can be specified as follows:

$$mergesort\,(x) \;=\; \begin{array}{l} base\,(x) \;\rightarrow\; sort(x)\,; \\ merge\;\,(mergesort\,(split_1(x))\,,\; mergesort\,(split_2(x))) \end{array} \tag{2}$$

Let us check that our example matches format (1). The function f to be computed is *mergesort*; predicate p, which tests the basic condition, is *base*. The simplest form of *base* is *single*, which is true if the sequence consists of just one element with the consequence that $b = id$. In the general case, the base case requires some computation: the sorting of x. Functions $split_1$ and $split_2$ generate two halves of the sequence, i.e. they implement the divide step of the algorithm. After these halves are sorted (recursively) by *mergesort*, function $E = merge$ combines them into a new ordered sequence. Thus, we distinguish three phases of divide-and-conquer: *divide*, *compute* and *combine*, applied in this order.

We view equation (1) as a specification for computing function f. Our goal is to generate a parallel program that, given a particular domain parameter *input*, computes $f(input)$ and, of course, all necessary values according to the dependences in (1). In our example, we are seeking a parallel implementation of (2) which must compute the value of $mergesort(input)$, where *input* is the input sequence.

Binary divide-and-conquer implies a virtual topology: a potentially infinite binary tree, whose leaves compute the base case. Because of the fixed processor number, we must embed this idealized topology into a finite one. If we do not want to leave the embedding up to the run-time system, we have basically two choices.

First, we can take the maximally possible *complete binary tree* of available processors [12]. Predicate *base* becomes true when instances cannot be divided further because of a lack of processors. We call such indivisible instances *tasks*; they are executed sequentially at the leaves of the tree; see Figure 1(a). Tasks are depicted as filled triangles which represent subtrees of the idealized processor tree. In the case of function *mergesort*, the tasks consist of sorting corresponding segments of the initial sequence in the leaf processors. Note that function *sort* is not necessarily the same as *mergesort* – one is free to use the optimal sequential algorithm in the tasks, e.g. the quicksort.

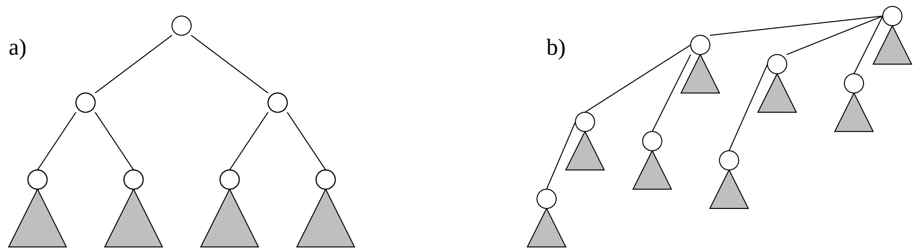

Figure 1: (a) Complete binary tree and (b) binomial tree

The drawback of the complete binary tree is that the load is not balanced. Only half of the processors are busy: $2^{(k-1)}$ of $2^k - 1$ for a tree of degree k. For large problems, where the main time is spent on computation in the leaves, this entails a 50% loss of speedup.

The problem of an unbalanced computational load is caused by a general property of most divide-and-conquer algorithms: they are *normal* [15] in the sense that only one level of the virtual binary tree is active at a time. For the divide and combine phase, normality means either that computations are performed at one level of the tree or that two neighbouring levels are communicating with each other. Therefore, the use of not only computational resources (processors) but also communicational resources (interprocessor links) is out of balance.

The second method of embedding exploits the property of normality as follows. Since the levels of the processor tree are active in a strict sequence and independently (except during communication between levels), each processor can be made responsible for one node at each level. Having performed the division, the processor keeps one of the two new instances and dispatches the other. This organization leads to the so-called *binomial tree* topology [16], shown in Figure 1(b), with tasks which are half as large as in the complete binary tree. In this case, the load is balanced better but we run into another problem: the node degree grows! At least the root in a binomial tree of degree k must have connections to k other nodes. A non-constant degree makes a topology non-scalable, which is a serious drawback. In the case of the transputer, which has 4 links, one can only build a binomial tree topology with up to 16 processors without sacrificing the locality of communication.

Our goal is to balance the computation load as well as the communication load. In particular, we would like to combine the advantages of both binary and binomial trees, i.e. obtain both a balanced load and a constant number of links per processor. This is achieved by introducing a new constant-degree topology.

3 $\mathcal{N}$-Graphs

In this section, we introduce $\mathcal{N}$-*graphs*, which are then used as a topology for a parallel implementation of divide-and-conquer. The number of nodes in an $\mathcal{N}$-graph is 2^k, for some $k \geq 2$. For a given k, we speak of a k-$\mathcal{N}$-graph. An $\mathcal{N}$-graph is directed, which allows us to speak of (direct) ascendants and descendants. However, when using an $\mathcal{N}$-graph as a processor topology, we interpret its edges as bidirectional links.

The base structure from which the topology takes its name is the 2-$\mathcal{N}$-graph shown in Figure 2. The *root* has no ascendants and the *leaves* have no descendants. We call the leaf node that is not a direct descendant of the root the *end* and the ascendant of the end the *bud*.

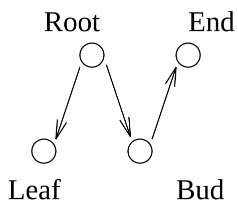

Figure 2: $\mathcal{N}$-graph : the base case

To obtain a k-$\mathcal{N}$-graph N, we combine two $(k-1)$-$\mathcal{N}$-graphs N_1 and N_2 (Figure 3). The end of N_1 becomes the root of N. We draw edges from it to the roots of N_1 and N_2. The end of N_2 becomes the end of N. To maintain planarity, we "pull" the root of N upwards and thus "stretch" the corresponding edge. The leaves and buds of N_1 and

N_2 become the leaves and buds of N. Nodes of an $\mathcal{N}$-graph which are not the root, the end, leaves or buds are called *inner* nodes.

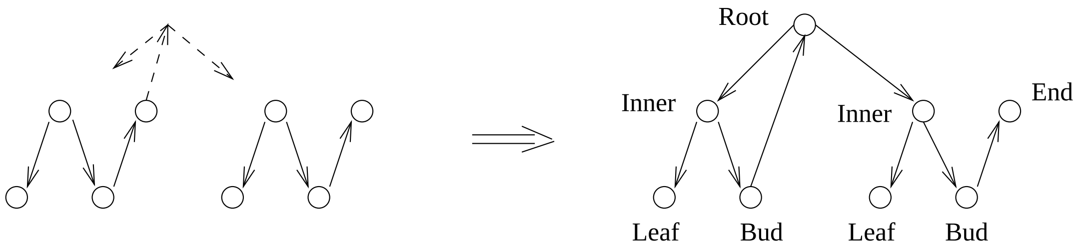

Figure 3: Construction of 3-$\mathcal{N}$-graph from two 2-$\mathcal{N}$-graphs

$\mathcal{N}$-graphs have the following properties.

Property 1. A k-$\mathcal{N}$-graph has 2^k nodes: one root, one end, $2^{(k-2)}$ leaves, $2^{(k-2)}$ buds and $2(2^{(k-2)}-1)$ inner nodes.

Property 2. All nodes in an $\mathcal{N}$-graph have a constant degree not greater than 4.

Property 3. A k-$\mathcal{N}$-graph contains a complete binary tree of degree k, which is obtained by excluding the end node and all outgoing edges of buds.

Property 4. Consider an arbitrary $\mathcal{N}$-graph. Let L be the set of leaves and B the set of buds, $A(L)$ the set of (direct) ascendants of the leaves and $D(B)$ the set of descendants of the buds. Then: (i) sets L, B, $A(L)$ and $D(B)$ are disjoint and constitute together the set of all nodes of the graph; (ii) there are bijections $L \overset{l}{\leftrightarrow} A(L)$ and $B \overset{b}{\leftrightarrow} D(B)$ such that, for nodes $x \in L$ and $y \in A(L)$, $x \overset{l}{\leftrightarrow} y$ iff y is the ascendant of x, and, similarly, for nodes $x \in B$ and $y \in D(B)$, $x \overset{b}{\leftrightarrow} y$ iff y is the descendant of x.

Let us now view both the complete binary tree and the $\mathcal{N}$-graph as processor topologies with bidirectional links, i.e. let us consider them as undirected graphs from now on. For arbitrary $x \in L \cup B$ and $y \in A(L) \cup D(B)$, we say that x is the $\mathcal{N}$-*father* of y and y is the $\mathcal{N}$-*son* of x. According to Property 4, the nodes of an $\mathcal{N}$-graph are divided into two disjoint sets: the $\mathcal{N}$-fathers and the $\mathcal{N}$-sons. Moreover, there is a direct link between each $\mathcal{N}$-father and its $\mathcal{N}$-son.

Our next step is the derivation of a parallel implementation for binary divide-and-conquer on the $\mathcal{N}$-graph topology. In order to keep the derivation process provably correct, we need a formal framework which is introduced in the next section.

4 Derivation

Divide-and-conquer is parallelized using formal transformations in the Bird-Meertens formalism (BMF) [17]. We use only a small subset of BMF here:

$[\alpha]$	the type of lists with elements from α;
$\circ$	backward functional composition;
id	the identity function;
$map\,f$	for an unary function f, $map\,f\,[x_1,\cdots,x_n] = [f\,x_1,\cdots,f\,x_n]$;
$<>$	Backus' FP *construction*: $< f_1,\cdots,f_n >\ x = [f_1\,x,\cdots,f_n\,x]$.

It is assumed that functional composition binds less strongly than function application.

Specification (1) can be reformulated in the Bird-Meertens notation:

$$f \;=\; p \to b\;;\; E \circ map f \circ < \varphi_1, \varphi_2 > \tag{3}$$

This form can be viewed as a skeleton which is specialized for our example (2) as follows:

$$mergesort \;=\; base \to sort\,;\; merge \circ map\,(\,mergesort) \circ <\,split_1, split_2 > \tag{4}$$

Note that the fact that we use BMF over lists does not mean that the basic type τ in our example (2) must be a list type. We use lists in BMF for reasoning about the control structure of the computations: trees and $\mathcal{N}$-graphs.

All constituting functions of the skeleton are treated as parameters, without using their specific properties. This guarantees that a parallel implementation derived for the skeleton will be correct independent of how the skeleton is filled in. On the other hand, problem-specific properties of the base predicate and function and also of the combining function can often be an additional source of optimization.

The higher-order functions *map* and $<>$ in (3) expose the intuitive parallelism of divide-and-conquer. Our aim is to map this parallelism efficiently onto a multiprocessor topology. Specification (1) is a special case of the mutually recursive specification presented in [12], with one recursive function and two recursive calls. To obtain a functional parallel implementation, we use the approach of abstract data type transformations [5]. In addition to the (initial) abstract type τ of the domain parameters, we introduce a concrete type whose elements are nodes of a processor topology. In [12], this topology is a complete binary tree, with the resulting task distribution as shown in Figure 1(a).

Let us now demonstrate that the $\mathcal{N}$-graph topology has the features which we pose as desirable in the introduction. The constant degree is given by Property 2. It remains to be shown how to distribute the tasks of divide-and-conquer in a balanced way among processors such that only processors connected via direct links communicate. We take an arbitrary $\mathcal{N}$-graph with the only requirement that the problem exceeds the size of the graph substantially, i.e. that there remain non-singleton segments to be sorted in each leaf of the tree. Actually, this requirement is not necessary, but it reflects the situation typical in practice and simplifies the presentation.

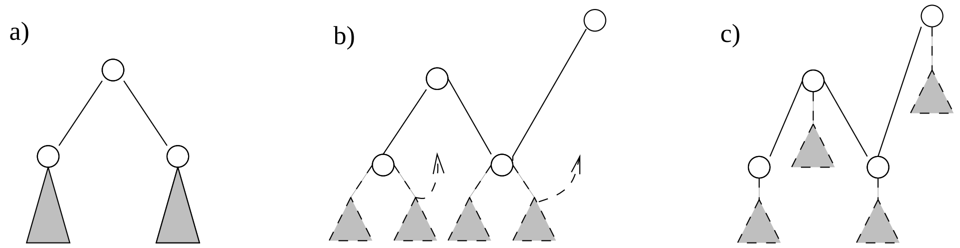

Figure 4: Distribution of tasks in $\mathcal{N}$-graph

Let us first look at a complete binary tree topology. We consider two leaves which have a common ascendant; see Figure 4(a). There is a task to be performed at each leaf. In an $\mathcal{N}$-graph, every task can be divided further into two subtasks. We can keep one of the subtasks in the corresponding leaf or bud (we call a node that is a leaf or a bud a *leaf-bud node*) and dispatch the other subtask to its $\mathcal{N}$-son; see Figure 4(b)-(c).

Property 4 guarantees that, after this transformation, every node of the $\mathcal{N}$-graph has exactly one subtask to execute.

This idea can be expressed formally in our functional setting. We introduce two groups of functions and predicates defined on the nodes of an $\mathcal{N}$-graph:

- The first group is defined on the complete binary tree which is a subgraph according to Property 3. Predicate $is.son_i$ indicates whether the node is the i-th son of some other node; predicates $is.root$ and $is.leaf$ have the obvious meaning expressed by their names; function son_i yields the i-th son of the argument node, and function $father$ yields the ascendant of the argument node.

- The second group of functions is $\mathcal{N}$-graph-specific: functions $nson$ and $nfather$ yield the $\mathcal{N}$-son ($\mathcal{N}$-father) of the argument node; predicates $is.leaf\text{-}bud$ and $is.end$ have the obvious meaning.

Now we have to consider two data types: the abstract type τ of the domain parameters and the concrete type of the nodes of an $\mathcal{N}$-graph. Let us construct an abstraction function that maps from the concrete to the abstract type, i.e. a function that assigns domain parameters to the nodes of the $\mathcal{N}$-graph.

According to the idea of subtask distribution depicted in Figure 4, each node receives two domain parameter values in the divide phase: the first from its father in the tree and the second from its $\mathcal{N}$-father. We express this by two abstraction functions called div_1 and div_2 (for "divide").

The abstraction functions are defined in the higher-order notation as follows:

$$\begin{aligned} div_1 &= is.end \to \perp\,;\ is.root \to input\,;\ (is.son_1 \to \varphi_1; \varphi_2) \circ div_1 \circ father\,; && (5)\\ div_2 &= is.leaf\text{-}bud \to \varphi_1 \circ div_1\,;\ \varphi_2 \circ div_1 \circ nfather && (6) \end{aligned}$$

Based on these abstraction functions, we introduce a new function defined on the nodes of the $\mathcal{N}$-graph:

$$F = < F_1\,, F_2 > = < f \circ div_1, f \circ div_2 > \qquad (7)$$

From (5), it follows that $F_1(root) = f(input)$, i.e. we have reduced our initial problem of implementing (1) by computing $f(input)$ to the problem of computing F_1 at the root of the $\mathcal{N}$-graph.

The computation of F_1 can be parallelized by formal transformations, leading to the following final expressions:

$$\begin{aligned} F_1 &= is.end \to \perp; && (8)\\ &\quad\ is.leaf\text{-}bud \to E \circ <f \circ \varphi_1 \circ div_1\,,\ F_2 \circ nson>\,;\ E \circ map\, F_1 \circ <son_1, son_2> &&\\ F_2 &= f \circ div_2 && (9) \end{aligned}$$

Formulae (8)-(9), together with (5)-(6), yield a parallel functional implementation of (1) as an $\mathcal{N}$-graph of processors with function $F =< F_1, F_2 >$ computed by each processor. F is common to all nodes, but it refers to the position of the node via predicates $is.leaf\text{-}bud$, $is.root$, etc. Thus, the behaviour of the processor depends on its position in the topology, i.e. we have the usual single-program-multiple-data (SPMD) model of parallelism. The correctness of the implementation is guaranteed by the semantically sound derivation in BMF.

5 Imperative Program

In this section, we describe how the derived implementation can be converted into an imperative parallel program with message passing.

In the divide phase, the computation of F_1 proceeds according to the usual divide-and-conquer template in the complete binary tree. The difference is in the compute phase: in the leaf-bud nodes, processors perform the following three steps shown in Figure 5:

(a) The leaf-bud nodes keep one subtask for themselves and send the other one to their $\mathcal{N}$-sons. These communications are shown with solid arrows in Figure 5(a).

(b) Nodes execute their subtasks, i.e. compute function F_2 sequentially, without further dispatching; active nodes are shaded in Figure 5(b).

(c) Results of computing F_2 are sent back to the $\mathcal{N}$-fathers, i.e., to the leaf-bud nodes, as indicated in Figure 5(c).

Then, the combine phase proceeds as usual to the higher levels of the graph along the edges of the corresponding complete binary tree.

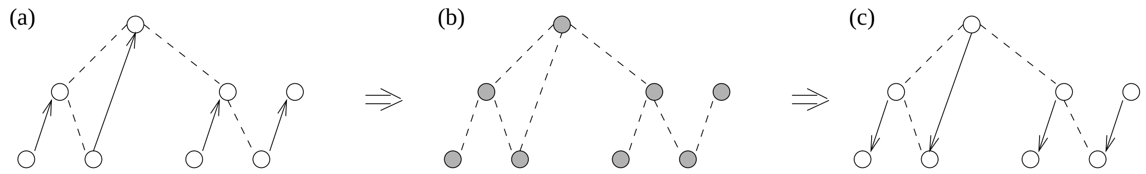

Figure 5: Active links and processors in the 3-$\mathcal{N}$-graph: (a) subtask distribution, (b) execution, (c) collecting the results

The transformation from the functional to the imperative representation requires explicit placement of computations and communications in the processors. We must decide e.g. whether the value of function div_1 is computed by each node itself or received from the father node.

We present the imperative program for our mergesort example in pseudocode with evident semantics; see Figure 6. Note that this program is just a specialization of the general binary divide-and-conquer schema (8)-(9) for the mergesort; we deliberately do not optimize it with reference to problem-specific properties. Let us look at how the constructs of the functional implementation are expressed imperatively.

Since there is a single function F to be computed at all nodes of the $\mathcal{N}$-graph, the expression for this function can be viewed as an SPMD program: our target program `Merge-Sort` is executed at every processor. One implicit parameter of program `Merge-Sort` is the identifier of the associated processor (variable `my_id`).

According to expression (7) of F, the program consists of two threads, one for F_1 and one F_2, which are executed in parallel due to the semantics of $<>$. Parallelism within a processor is expressed using the statement `PARBEGIN || PAREND`. A thread can be terminated with `BREAK`: this happens e.g. when the corresponding function is undefined according to (8)-(9). The thread corresponding to F_1 consists itself of two parallel threads due to $<>$ in (8).

```
PROGRAM Merge-Sort (my-id);
 IMPORT Father, Son, N_Father, N_Son, Is_leaf-bud, Is_root Is_end,
        Sort, Merge, Split_1, Split_2.
   PARBEGIN
    { IF Is_end(my_id) THEN BREAK END_IF;
      IF Is_root(my_id) THEN READ (par)
       ELSE RECV (par) FROM Father(my_id) END_IF;
      PARBEGIN
       { par[1] := Split_1 (par);
         IF Is_leaf-bud THEN
          res[1] := Sort (par[1])
           ELSE { SEND par[1] TO Son(my_id,1);
                  RECV res[1] FROM Son(my_id,1) } END_IF }
      || { par[2] := Split_2 (par);
           IF Is_leaf-bud THEN partner := N_Son(my_id)
            ELSE partner := Son(my_id,2) END_IF;
           SEND par[2] TO partner;
           RECV res[2] FROM partner }
      PAREND;
      result := Merge (res[1],res[2]);
      IF Is_root (my_id) THEN WRITE (result)
       ELSE SEND (result) TO Father (my_id) END_IF }
   || { IF (Is.leaf-bud) THEN BREAK;
        RECV (parn) FROM N_Father(my_id);
        resn:= Sort (parn);
        SEND (resn) TO N_Father(my_id) }
  PAREND;
END_Merge-Sort;
```

Figure 6: SPMD program

Each processor has its own local memory. In the functional representation, functions receive parameters and transfer results implicitly between each other. In the imperative program, we use variables and assignments for this purpose. Variable `par` stores the domain parameter for the current processor. Since the domain parameter for a node is computed by its father, we need also an array `par[]` for storing the corresponding values computed for the sons. Array `res` contains the ordered sequences received from the sons. Finally, `result` stores the value of F at the current node. Computations of functions f, φ_1 etc. are implemented by procedures with corresponding names: `Sort`, `Split_1`, etc.

Input-output is implemented by the statements `READ` and `WRITE`. The root node inputs the input argument and outputs the solution to the complete problem.

Processors send and receive data by the statements `SEND (<data>) TO <partner>` and `RECV (<data>) FROM <partner>`. Here, `<partner>` is the id of the processor with whom the communication takes place. For the sake of simplicity, we consider synchronous blocking communication; sometimes asynchronous communication can yield better performance.

6 Generalization of the $\mathcal{N}$-graph

There are important divide-and-conquer algorithms that use not two, as in the binary case, but three or more recursive calls: e.g. an optimized algorithm of polynomial multiplication has dividing factor 3 [18]. In principle, the $\mathcal{N}$-graph topology can be generalized to an arbitrary dividing factor.

The $\mathcal{N}$-graph topology for $d=3$ and $k=2,3$ is shown in Figure 7; leaf-bud nodes are shaded.

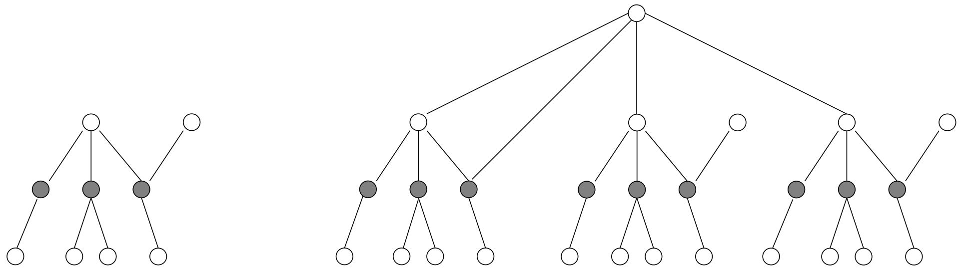

Figure 7: 2- and 3-$\mathcal{N}$-graph for ternary divide-and-conquer

The four properties of the $\mathcal{N}$-graph topology for divide-and-conquer with an arbitrary dividing factor d can be formulated as follows:

1. $k-\mathcal{N}$-graph has d^k nodes, $k \geq 2$.

2. The maximal degree of nodes is $d+2$.

3. The complete d-ary tree is a subgraph of the topology.

4. There are d^{k-1} leaf-bud nodes in the graph; $d^{k-1} \cdot (d-1)$ other nodes can be partitioned into d^{k-1} subsets, one for each leaf-bud, such that all $d-1$ nodes in the subset are directly connected with the corresponding leaf-bud.

It should be noted that, for $d>2$, the number of extra nodes in the $\mathcal{N}$-graph grows so quickly compared with the complete tree of degree d, that the advantages of the $\mathcal{N}$-graph are destroyed by its additional resources requirements. Already in the case of ternary divide-and-conquer, the number of extra nodes is $n+1$, where n is the number of nodes in the corresponding complete ternary tree. In other words, we need roughly twice as many processors.

In the case of the binary divide-and-conquer, we need only one additional node, regardless of the size of the graph. This node is usually already present in the topology, because the total number of nodes is a power of 2 in most multiprocessor systems: we just take the processor which otherwise would be idle in the complete binary tree topology.

Therefore, we believe that the $\mathcal{N}$-graph topology is practical only for binary divide-and-conquer.

7 Transputer Implementation and Experimental Results

7.1 Implementation Platform

We have studied the performance of the presented parallel implementation of the mergesort on a 64-node *Parsytec* transputer system. The nodes of the system are T805 transputers which are physically connected in a two-dimensional grid. They run the Parix operating system [19], which provides a set of functions for building and using communication topologies matching particular problems and for defining suitable routines managing data operations. Parix offers different types of synchronous and asynchronous communication. We use a library of virtual topologies provided by Parix, which relieves the programmer from having to build communication networks for some standard problems.

In order to exploit the advantages of the $\mathcal{N}$-graph topology fully, it would be desirable to realize it physically, e.g. by configuring the necessary links electronically, which is possible in some transputer systems. Due to the lack of access to such a system, both the complete binary tree and the $\mathcal{N}$-graph topology were implemented as virtual topologies in Parix.

The pseudocode of Figure 6 converts easily to our programming language: C with point-to-point communication. The parallel statements `PARBEGIN || PAREND` can be implemented as a set of parallel threads.

7.2 Implementation on the tree

In order to gain a better understanding of the influence of various optimizations on the performance of divide-and-conquer, we separated the three phases of our algorithm and measured the run time for each of them.

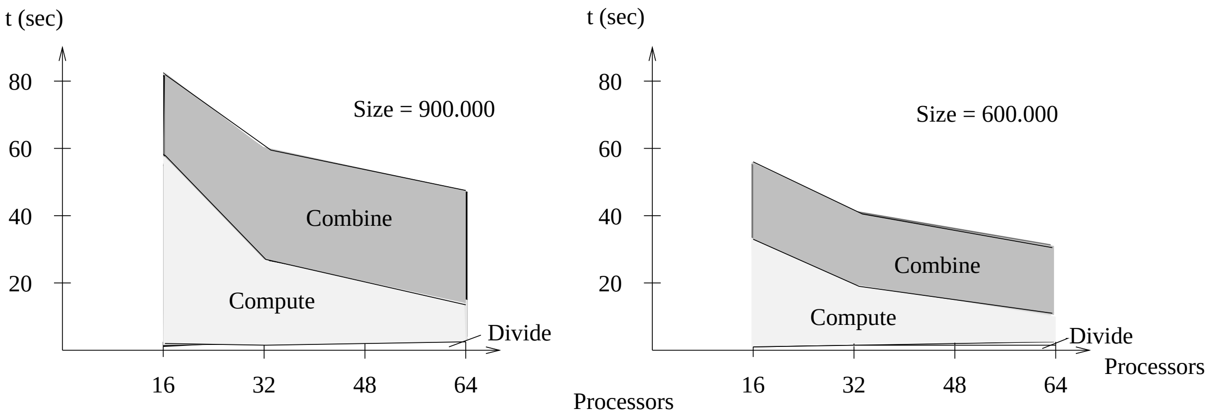

Figure 8: Complete binary tree topology: run times

The two plots in Figure 8 show the total run time of mergesort on a complete binary tree and the contributions of the phases of divide-and-conquer for two problem sizes (lengths of the sorted sequence). It is clear from the figure that the compute and combine phase are the candidates for possible optimization. The centralized input of data, which is often seen as a source of inefficiency in divide-and-conquer, plays a minor rôle here.

7.3 Implementation on the $\mathcal{N}$-graph: compute phase

As described in Section 4, the main purpose of the $\mathcal{N}$-graph topology is to optimize the compute phase of divide-and-conquer. There is also another opportunity for optimization at the transition between the the compute and the combine phase. Each leaf node can send its result immediately to its father (of whom it is the left son) and terminate; the father combines the received data with the result it has computed itself. At the same time, the buds (which are the right sons of their fathers) receive results from their $\mathcal{N}$-sons and combine. It remains then for the father to receive the result from its right son and to combine.

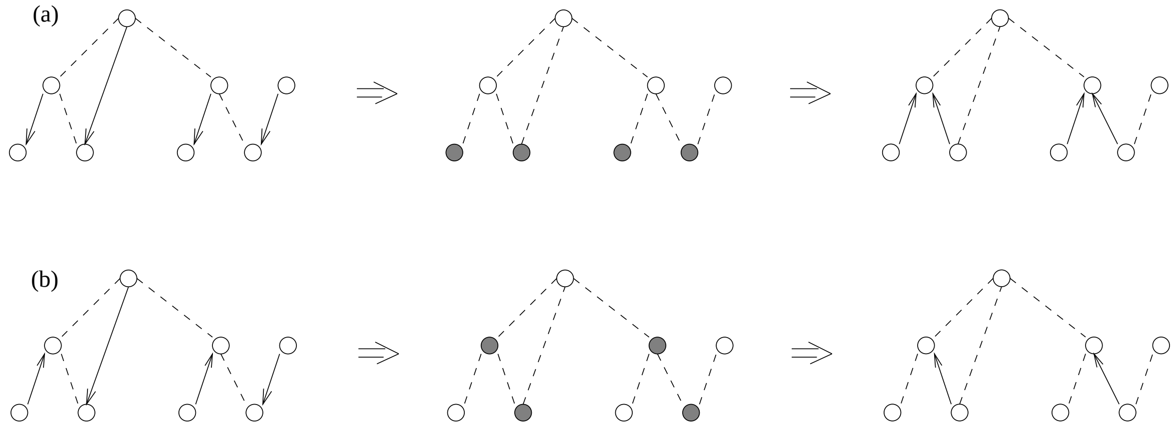

Figure 9: Collecting and combining results: (a) before and (b) after optimization.

This optimization is shown in Figure 9; nodes which are combining results are shaded. The optimization saves one communication of each leaf node with its father. We have implemented this version in our experiments.

The plots in Figure 10 depict how the time of the compute phase changes when the $\mathcal{N}$-graph topology is used in place of the complete binary tree.

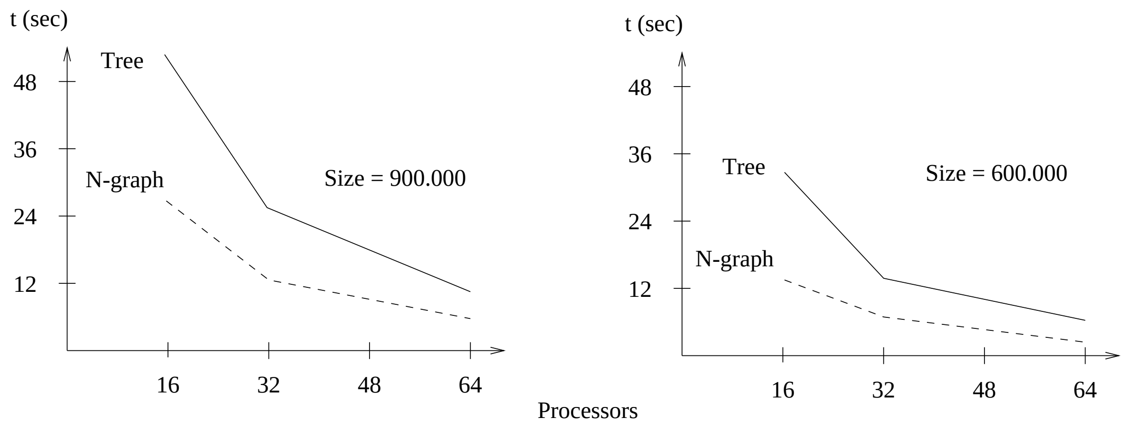

Figure 10: Compute times for the complete tree and the $\mathcal{N}$-graph

7.4 *Combine phase*

The improvement due to the $\mathcal{N}$-graph influences only the two lowest levels of the virtual binary tree. At all other levels, both the divide and combine phase remain normal, i.e. each level is executed only after the previous has terminated. The consequence becomes evident in Figure 8, where the combine phase constitutes, in some cases, more than half of the total run time. Therefore, this phase is our next goal of optimization.

The first possibility to improve the combine phase is to use asynchronous communication as provided in Parix, exploiting the ability of the transputer to overlap communication on different links. Thus, each node can receive data from both sons simultaneously. In fact, this organization is exactly what is expressed by the `PARBEGIN-PAREND` parentheses in Figure 6. A drawback of asynchronous communication is that it requires additional memory for buffers.

Another possibility for optimization of the combine phase is to pipeline it. The idea of pipelining is quite obvious: instead of receiving, merging and sending the entire ordered sequence, a processor divides it into segments and processes each of them consecutively. Note that we do not need any additional memory for organizing the pipelined execution. An additional point of optimization could be to use the feature of overlapping of computation and communication in a transputer.

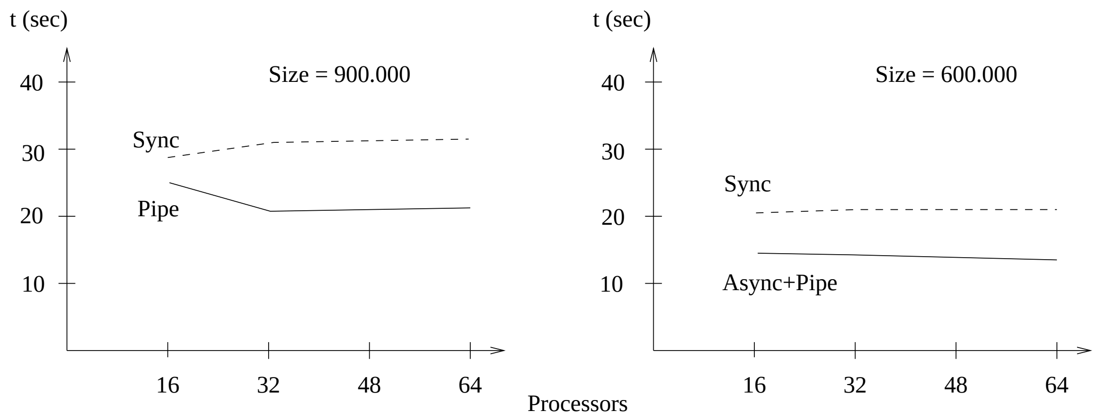

Figure 11: Combine time optimization

We have studied the run time of the combine phase for the three communication modes described above: synchronous, asynchronous and pipelined. Corresponding plots are shown in Figure 11. The best performance is gained when both pipelining and asynchronous communication are exploited. Asynchronous communication could not be implemented for problem size 900.000 because of the induced additional memory requirements.

7.5 *Speed-up*

As the cumulative result of all optimizations described so far, Figure 12 depicts the speed-up curves achieved by the complete tree and the $\mathcal{N}$-graph-topology for the same problem size. To compute the speed-up, we compare our parallel mergesort algorithm with the best sequential algorithm (quicksort) implemented on one transputer.

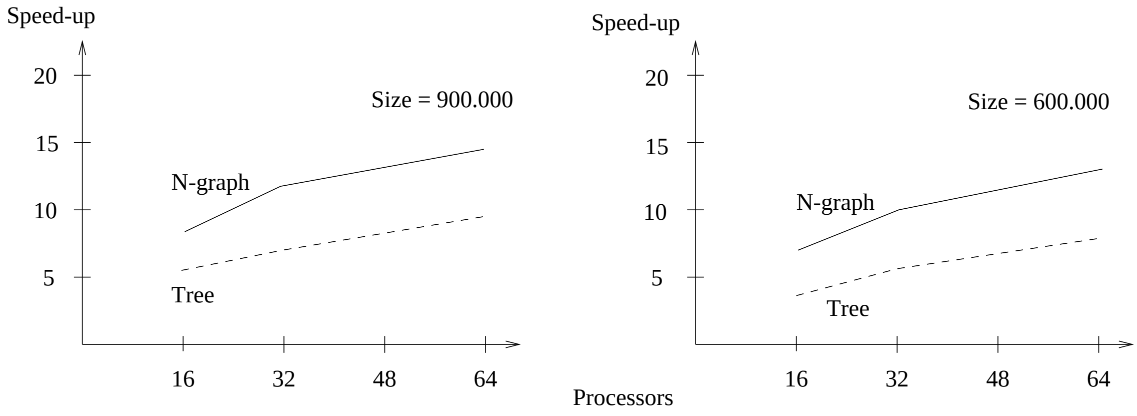

Figure 12: Speed-ups

The plots are intended to give an impression of the behaviour of the speed-up values on the $\mathcal{N}$-graph as opposed to the complete binary tree. The absolute figures are comparatively modest because both topologies have been embedded into a two-dimensional mesh. It is known that the dilation of such an embedding is proportional to the square root of the processor number. A physical implementation of the $\mathcal{N}$-graph, in which all neighbour nodes are connected directly, should improve the absolute performance substantially.

8 Conclusion

Divide-and-conquer is one of the widely used templates of parallelism. There are two main problems with its practical implementation on multiprocessor networks.

The first problem (we have mentioned it in the introduction and addressed it in this paper) is how to map the problem instances onto a constant-degree processor network of a fixed size with balanced load and local communication. We have solved this problem by introducing a new topology with degree 4: the $\mathcal{N}$-graph. This topology fits nicely on transputer networks. A parallel implementation of divide-and-conquer on the $\mathcal{N}$-graph has been derived in a provably correct way: by calculation from the initial functional specification using the semantically sound rules of the Bird-Meertens formalism.

Our experiments have shown that, even when simulated on some other physical topology, $\mathcal{N}$-graphs allow for considerable improvements of performance. In the compute phase, $\mathcal{N}$-graphs yield a speed-up 2, compared with the complete binary tree. While this is an upper bound on the expected improvement (in the binary case), it provides, in practice, reasonable gains at low cost.

The second problem is that the time of dividing and combining data may exceed the computation time in the leaves which makes divide-and-conquer "inherently inefficient" [20]. We have demonstrated that, even in this case, there are still opportunities for an optimization, such as pipelining. In our parallel implementation, the number of processors can be chosen independently of the problem size. The only restriction is that this number must be 2^k for $k \geq 2$.

A limitation of our approach is that it is restricted to the class of binary balanced algorithms which, despite their wide applicability, do not cover the whole class of divide-and-conquer algorithms.

Our implementation covers the format of specification (1). Further optimizations can be obtained by exploiting problem-specific properties.

9 Acknowledgment

Benoît Caillaud drew our attention to the possibility of generalizing $\mathcal{N}$-graphs to dividing factors higher than 2.

References

[1] M. Cole. Algorithmic skeletons: A structured approach to the management of parallel computation. Ph.D. Thesis. Technical Report CST-56-88, Department of Computer Science, University of Edinburgh, 1988.

[2] J. Darlington, A. Field, P. Harrison, P. Kelly, D. Sharp, Q. Wu, and R. While. Parallel programming using skeleton functions. In A. Bode, M. Reeve, and G. Wolf, editors, *Parallel Architectures and Languages Europe, PARLE '93*, Lecture Notes in Computer Science 694, pages 146–160. 1993.

[3] Z. G. Mou. Divacon: A parallel language for scientific computing based on divide and conquer. In *Proc. 3rd Symposium on the Frontiers of Massively Parallel Computation*, pages 451–461, October 1990.

[4] Z. G. Mou and P. Hudak. An algebraic model for divide-and-conquer algorithms and its parallelism. *Journal of Supercomputing*, 2(3):257–278, 1988.

[5] P. G. Harrison. A higher-order approach to parallel algorithms. *The Computer Journal*, 35(6):555–566, 1992.

[6] I. P. de Guzman, P. G. Harrison, and E. Medina. Pipelines for divide-and-conquer functions. *The Computer Journal*, 36(3):254–268, 1993.

[7] D. McBurney and M. Sleep. Transputer-based experiments with the ZAPP architecture. In J. de Bakker, A. Nijman, and P. Treleaven, editors, *PARLE Parallel Architectures and Languages Europe*, Lecture Notes In Computer Science 258, pages 242–259. 1987.

[8] S. Cox, S. Huang, P. Kelly, J. Liu, and F. Taylor. An implementation of static functional process networks. In D. Etiemble and J.-C. Syre, editors, *PARLE'92 Parallel Architecture and Languages Europe*, Lecture Notes in Computer Science 605, pages 497–512, 1992.

[9] P. Pepper. Deductive derivation of parallel programs. In R. Paige, J. Reif, and R. Wachter, editors, *Parallel Algorithm Derivation and Program Transformation*, pages 1–53. Kluwer Academic Publishers, 1993.

[10] Z. G. Mou, C. Constantinescu, and T. J. Hickey. Divide-and-conquer on three-dimensional meshes. In W. Joosen and E. Milgrom, editors, *Parallel Computing: From Theory to Sound Practice*, pages 344–355. IOS Press, 1992.

[11] A. K. Gupta and S. E. Hambrusch. Load balanced tree embeddings. *Parallel Computing*, 18:595–614.

[12] S. Gorlatch and C. Lengauer. Systematic development of an SPMD implementation schema for mutually recursive divide-and-conquer specifications. In H. J. Siegel, editor, *Proc. Eighth Int. Paral. Proc. Symp. (IPPS'94)*, pages 368–375. IEEE Computer Society Press, 1994.

[13] S. Gorlatch. Formal derivation and implementation of divide-and-conquer on a transputer network. In A. De Gloria, M. Jane, and D. Marini, editors, *Transputer Applications and Systems '94*, pages 763–776. IOS Press, 1994.

[14] J. W. Backus. Can programming be liberated from the von Neumann style? *Communications of the ACM*, 21:613–641, 1978.

[15] F. T. Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes.* Morgan Kaufmann Publ., 1992.

[16] M. J. Quinn. *Parallel Computing.* McGraw-Hill, Inc., 1994.

[17] R. S. Bird. Lectures on constructive functional programming. In M. Broy, editor, *Constructive Methods in Computing Science*, NATO ASO Series F: Computer and Systems Sciences. Vol. 55, pages 151–216. Springer Verlag, 1988.

[18] R. Sedgewick. *Algorithms.* Second edition, 1988.

[19] Parix. *Release 1.2. Software Documentation.* Parsytec Computer GmbH, 1993.

[20] P. B. Hansen. Model programs for computational science: A programming methodology for multicomputers. *Concurrency: Practice and Experience*, 5:407–423, aug 1993.