TPascal - A Language for Task Parallel Programming

Ansgar Brüll and Herbert Kuchen

Lehrstuhl für Informatik I/II, RWTH Aachen, D-52056 Aachen, Germany
email: {bruell,herbert}@i2.informatik.rwth-aachen.de

Abstract. The programming language *TPascal* is designed for the programming of MIMD computers with distributed memory in a task parallel way using explicit message passing. In contrast to traditional message passing libraries major problems like deadlocks are already avoided in the definition of the language. The message passing and the creation of processes is fully integrated into the language making compile time checking and optimization possible. *TPascal* enriches a sequential programming language similar to *PASCAL* with the concept of *topologies* which are sets of processes arranged in a specific way.

1 Introduction

Programming MIMD computers with distributed memory is still difficult, error prone and time consuming. Mostly, traditional sequential programming languages together with a message passing library like *P4*, *PVM* or *MPI* (see [2] for references) are used. Such library calls allow only very limited compile-time checking and optimization of the communication. Various of the mentioned libraries transmit data as strings making even run-time checking impossible.

In contrast to the explicit use of message passing routines, different languages have been developed achieving a higher degree of abstraction from the underlying system, e.g. *High Performance Fortran (HPF)* [5]. The programmer can distribute data over virtual SPMD processes and perform data parallel operations on arrays of arbitrary shape. All communication statements necessary to execute such programs on truly parallel computers are inserted by the compiler. Deadlocks cannot occur.

While the number of data parallel applications is quite large, many applications and algorithms can be better thought of as a number of independent processes connected and communicating in a specific way. This *task parallel* paradigm has been used as the basis of a number of languages. E.g. *SVM Fortran* [1] uses parallel sections and parallel loops where each section and each loop iteration is an independent process. Synchronization and data exchange between the processes can be achieved by global data that has to be shared between the processes and is realized by means of virtual memory. To regulate the data access the programmer has to make use of standard low level synchronization methods like semaphores and atomic updates making deadlocks possible. The approach taken by *Fortran M* [6] is based on explicit message passing. The programmer

can declare processes which can exchange data through typed channels. These channels connect two arbitrary processes, again enabling deadlocks. A more static approach for the communication has been chosen within *FX* [7], where tasks can communicate only through arguments at the time of creation and termination. The advantage of all these languages over the traditional message passing libraries is that the data exchange protocol can be type checked by the compiler. However, programs may still run into deadlocks.

To avoid them, *TPascal* takes an approach quite similar to *skeletons* [4] in functional programming. A skeleton can be regarded as a parameterized template of a specific parallel operation. The implementation of a skeleton is hidden from the programmer and therefore the programmer does not need to tackle the 'low level problems' of parallelism. Within *TPascal* these templates are represented by *topologies* which are sets of processes being connected to each other in a particular way, e.g. pipes, rings etc. Topologies can be nested enabling the construction of large collections of processes. A similar scheme has been used in P^3L [3] where the result of a computation is transferred to a different process by variables that are shared between the processes. *TPascal* however, uses explicit message passing. Only processes being connected can exchange data in a well defined, topology dependent way. For each topology only certain ways for a data exchange exist, making the occurrence of deadlocks impossible. Through the integration of the primitives into the language the compiler can perform many optimizations. Additionally, the data exchange protocol between two connected processes can be verified by the compiler and the runtime system with respect to the types of the data being exchanged.

2 A Description of TPascal

TPascal consists of two parts. The first one, the *host language*, is made up of a sequential programming language containing usual constructs like loops, conditionals etc. The second component is the *coordination part* of the language consisting of constructs to set up parallel processes and to exchange data between them. Let us first consider an example.

The program in Fig. 1 computes the sum of an array of integers. It consists of a ring of five processes. Each process initializes its local sum by an argument received at its creation. Then, it stores the local sum into two auxiliary variables, SUM1 and SUM2, which are then used to exchange these values with the neighbor processes. The EXCHANGE_NB operation has for each neighbor an argument list telling what to send to (OUT) and what to receive from this neighbor (IN). The received values are added to the local sum, and then passed to the opposite neighbor in the next step. Thus, in the i-th iteration, the initial local value reaches the neighbors with distance i.

2.1 The Coordination Language

The constructs from the coordination language are used for expressing parallelism within a program by setting up parallel processes and for exchanging data

```
PROGRAM ADD
VAR Data : ARRAY [1..5] OF INTEGER;

PROCEDURE Calc(A: INTEGER)
VAR I,SUM,SUM1,SUM2:INTEGER;
BEGIN
   SUM:=A; SUM1 := SUM; SUM2 := SUM;
   FOR I:=1 TO 2 DO
   BEGIN
      EXCHANGE_NB([IN SUM1, OUT SUM2],[OUT SUM1, IN SUM2]);
      SUM := SUM + SUM1 + SUM2;
   END
END

BEGIN
   /* Start of the main program, Initialization of array Data, ...*/
   RING(Calc(Data[1]), Calc(Data[2]), Calc(Data[3]), Calc(Data[4]), Calc(Data[5]));
END
```

Fig. 1. A simple TPascal program.

between these processes.

Parallelism is expressed by explicitly creating *topologies*, which are sets of parallel processes called *tasks* connected in a specific way. An atomic task is a procedure call started as a separate process with its own address space. Topologies are established by constructs consisting of a keyword describing the connection and a corresponding list of tasks. The constructs are integrated into the host language as statements. The interconnections between the tasks within a topology determine the way of exchanging data. Only connected tasks can communicate. The programmer can choose between different predefined topologies.

Communication Primitives. Depending on the topology certain tasks are connected and can exchange data by two primitives. To exchange data with all connected tasks (called *neighbor tasks*), EXCHANGE_NB can be used. These tasks are referred to as. A non-empty list of expressions has to be supplied to the primitive for each connected task together with an attribute that specifies whether the data will be received from (IN attribute) or send (OUT attribute) to a connected task. Another operation EXCHANGE (only available for some topologies) allows to communicate with one selected neighbor. Implementation aspects are discussed in [2].

Creating Topologies. In *TPascal* constructs are available for the most convenient topologies like parallel independent tasks, rings of tasks, pipelines of tasks etc. (see Fig. 2). For each type of topology a construct is available that has as parameter a list of topologies together with their actual parameters. When a new topology is created by executing a task construct, the parts of the state of the starting task needed for the new tasks are duplicated. Within *TPascal* tasks will not necessarily be executed on the same physical processor in which case the runtime system is responsible for creating an initial state on the processors

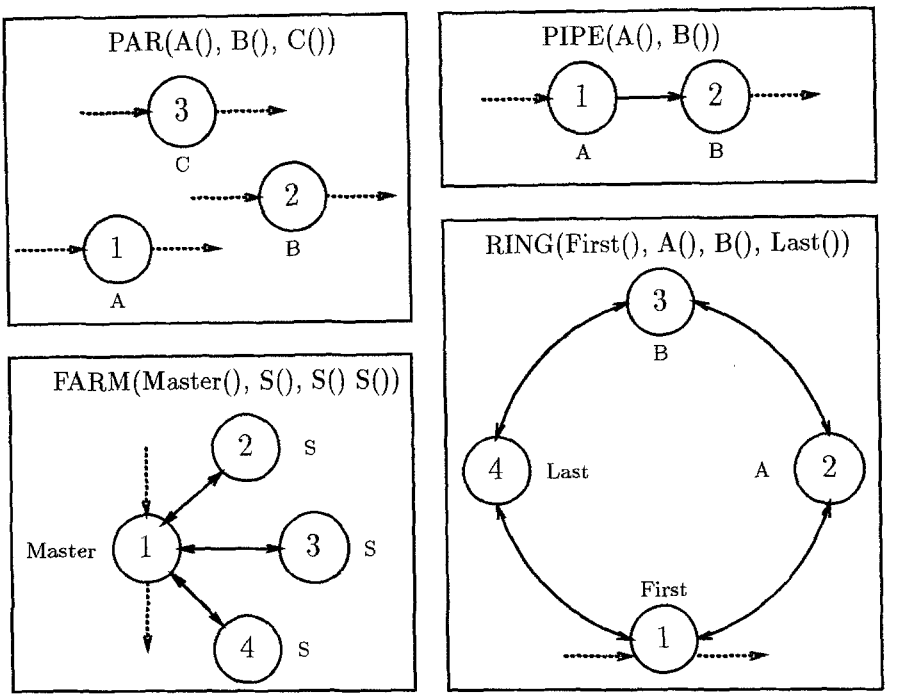

Fig. 2. Topologies: parallel independent processes, pipe, farm, and ring.

involved in that operation. Procedure calls started as tasks can, of course, contain further topology constructs. A topology construct terminates when all tasks within the topology have terminated.

The simplest topology consist of parallel independent tasks (created by the PAR construct), which cannot communicate with each other. In Fig. 2, the tasks are represented by circles labeled with some internal task number needed for the connection of different topologies. All tasks that have an incoming dotted arrow expect some input from previous topologies, while all tasks with an outgoing dotted arrow will send some output to subsequent topologies. The previous and subsequent topologies are determined by a topology construct at an outer level of nesting (explained below).

The PIPE construct enables the programmer to define a unidirectional sequence of not necessarily equal tasks. Ordinary SEND and RECEIVE instructions may be used, since a pipe is acyclic and there is no danger of deadlocks.

The ring topology allows a cyclic exchange of data in a bidirectional way. All communication that is performed between two tasks being connected by a RING construct has to be done by the EXCHANGE_NB primitive.

A farm allows one master to send jobs to a number of slaves which in turn work on these jobs and return intermediate results, final results and possibly new jobs. This process of distributing jobs to the slaves continues until no more jobs are available.

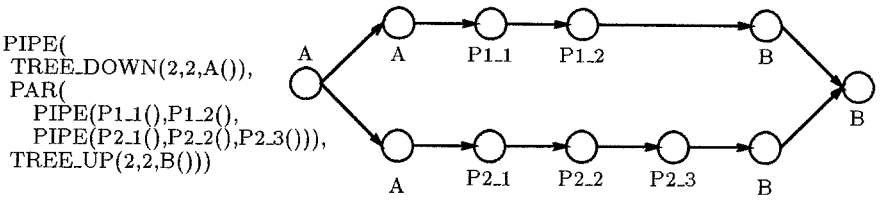

Fig. 3. Nesting topologies

The EXCHANGE_NB primitive can be used to ensure that the slaves work synchronously. The EXCHANGE operation allows to give jobs to selected slaves and thus to get an asynchronous behavior of the farm. The master may also use an EXCHANGE operation, where the desired communication partner is left open. In this case, the master communicates with the first ready slave. This is the only place, where non-determinism occurs.

The tree is a generalization of the pipe. It can be regarded as a branching pipe. We distinguish trees with a data flow from the root to the leaves (TREE_DOWN), and trees with the opposite data flow (TREE_UP). The above topologies can be nested. An example of a nested topology is shown in Fig. 3.

Combining Topologies with Constructs of the Host Language. The topologies described are static with respect to their communication structure. By using the topology primitives in combination with the control structures of the host language, tasks can be established dependent on the state of program execution e.g. within conditionals or loops. Tasks can be created recursively when used in connection with recursive procedures. Thus, in combination with the constructs of the host language a dynamic process structure can be established. Tasks started in different iterations of a loop or in different levels of a recursive procedure can only "communicate" via the host language. Certain features of the host language have to be handled in a restricted way, e.g. pointers and I/O. See [2] for details.

2.2 Deadlock Avoidance in TPascal

A necessary condition for a deadlock to occur is that a circular dependence has to exist between the processes. By induction on the nesting level, it can be shown that no deadlock can ever occur in a *TPascal* program. In the following we present a rough sketch of the induction step of this proof. For the induction hypothesis we assume the topologies $T_1, \ldots, T_n$, that are used for the construction of a new topology T, to be deadlock free. If T is a pipe or a tree topology, it is obvious that no deadlock can ever occur, as no cyclic dependences are introduced by these constructs. If T is a farm, the data exchange always takes place in a bidirectional way within a single primitive. Therefore no circular dependencies can occur. For a ring topology, a similar argument holds.

It should be noted that other forms of congestion besides deadlocks are still possible. This is mainly due to the fact that their avoidance within the language definition would impose large restrictions on the host language making it impractical. Among these problems are *livelocks* where a number of tasks wait for an event that is never going to happen. In particular, an infinite loop can prevent a process from reaching its next communication instruction. This will cause the communication partners to wait forever.

3 Summary and Discussion

We have presented *TPascal*, a language for programming distributed memory MIMD machines based on the task parallel paradigm using explicit message passing. Tasks are arranged in topologies. The topology determines which tasks can directly exchange data. *TPascal* simplifies programming as deadlocks within user programs are avoided by restricting the communication. Both the primitives for task creation and communication are part of the language enabling the compiler to perform necessary optimizations. Another feature of *TPascal* is its abstraction from the underlying hardware. The programmer can choose between a given set of topologies like trees, pipes, farms and parallel independent tasks. A mapping of the tasks that form a topology to the physical processors of the target machine is done by the runtime system in a way that the routing of messages between tasks is done on the shortest way possible and the load is equally distributed onto the physical processors available. This is possible as the way tasks exchange data is known in advance through the use of the topologies with only a fixed way of exchanging data. In the future, we want to extend *TPascal* by data parallelism. This will be done by some skeletons (topologies) operating on a distributed data structure.

References

1. R. Berrendorf, M. Gerndt, W. Nagel, and J. Prümmer. SVM-Fortran. Report KFA-ZAM-IB-9322, Forschungszentrum Jülich, 1993.
2. A. Brüll, H. Kuchen. TPascal - A Language for Task Parallel Programming. Report, `http://www-i2.informatik.rwth-aachen.de/~herbert/TRGiBrKu.ps`
3. M. Danelutto, R. DiMeglio, S. Orlando, S. Pelagatti, and M. Vanneschi. A methodology for the development and the support of massively parallel programs. *Future Generation Computer Systems*, 205 – 220. Elsevier, 1992.
4. J. Darlington et al. Parallel Programming Using Skeleton Functions. PARLE'93, LNCS 694, Springer, 1993.
5. The High Performance Fortran Forum. *High Performance Fortran Language Specification*, 1993.
6. I.T.Foster and K.M. Chandy. *Fortran M: A Language for Modular Parallel Programming*, June 1992.
7. J. Subhlok and T. Gross. Task parallel programming in FX. Report CS-94-112, Carnegie Mellon University, 1994.