A Tool for Generating Programs with Mixed Task and Data Parallelism

Dissertation

zur Erlangung des akademischen Grades
doctor rerum naturalium (Dr. rer. nat.)

vorgelegt der

Mathematisch-Naturwissenschaftlich-Technischen Fakultät
(mathematisch-naturwissenschaftlicher Bereich)
der Martin-Luther-Universität Halle-Wittenberg

von Ursula Fissgus
geb. am 08. Juli 1957 in Lupeni

Gutachter:

1. Prof. Dr. Thomas Rauber
2. Prof. Dr. Gudula Rünger
3. Prof. Dr. Wolf Zimmermann

Halle(Saale), 08. Juni 2001

Abstract

Distributed memory machines provide large computing power, but the development process for a specific parallel algorithm on a specific machine is complex due to the complicated runtime behaviour caused by communication overhead and load imbalance. We consider a powerful multi-dimensional scheduling embedded into a tool for generating parallel programs with mixed task and data parallelism.
The program generation starts with a user provided specification of the maximum degree of task and data parallelism of the method to be implemented. In several derivation steps, the system adapts the degree of parallelism to a specific target machine. The therefore used scheduling is based on the genetic algorithm paradigm. Our scheduling takes not only decisions on the execution order (independent modules can be executed consecutively by all processors available or concurrently by independent groups of processors) and on the mapping of processors to tasks, but also on appropriate data distributions and module implementation versions (for each module there are several implementation version available, e.g., taken from a predefined set of library functions). Data redistribution operations and communication domain management operations are added, if necessary. The obtained parallel frame program can be translated in any imperative language augmented by a message passing library supporting groups.
The efficiency of our system was demonstrate by several examples.

Zusammenfassung

Parallelrechner, insbesondere solche mit verteiltem Speicher, bieten eine hohe Rechenkapazität, weisen aber eine komplexe Programmierbarkeit auf. Die vorliegende Arbeit beschreibt ein Werkzeug zur Generierung von parallelen Programmen mit gemischter Daten- und Funktionsparallelität auf Rechner mit geteiltem Speicher. Das Werkzeug stützt sich auf ein vielseitiges multidimensionales Scheduling, welches ebenfalls in der vorliegenden Arbeit beschrieben wird.
Die Programmgenerierung geht von einer vom Benutzer angegebenen Spezifikation der zu implementierenden Anwendung aus. Diese Spezifikation beschreibt die potentiell vorhandene Parallelität der Anwendung, der anschließend von unserem System in mehreren Ableitungsschritten an den Gegebenheiten eines Zielrechners angepasst wird. Das von uns entwickelte Scheduling, das auf genetische Algorithmen basiert, entscheidet nicht nur über die Reihenfolge der Abarbeitung (voneinander unabhängige Module können einer nach dem anderen von allen Prozessoren datenparallel abgearbeitet werden oder sie können gleichzeitig, funktionsparallel, von disjunkten Gruppen von Prozessoren abgearbeitet werden) und darüber welche Prozessoren den einzelnen Modulen für die Abarbeitung zugeordnet werden, sondern trifft auch Entscheidungen über die geeignete Datenverteilung und über die jeweils einzubindende Implementierungsversion (dem System können für einzelne Module mehrere Implementierungen, zum Beispiel unterschiedliche Bibliotheksfunktionen, zur Auswahl vorliegen). Datenumverteilungsoperationen und Operationen zur Verwaltung der Kommunikationsumgebung werden, bei Bedarf, dem erstellten parallelen Rahmenprogramm hinzugefügt. Das Rahmenprogramm wird unter Verwendung einer imperativen Programmiersprache und einer Kommunikationsbibliothek, die hierarchische Prozessorgruppenstrukturen unterstützt, in einem parallelen Programm umgewandelt.
Die Einsatzfähigkeit und Effizienz unseres Ansatzes wurde an verschiedenen Beispielen erprobt.

Contents

1 Introduction

Massively parallel systems promised to increase computing performance by several orders of magnitude. The recognition of this fact leads to intensified research in the field of parallel and distributed system in the eighties. Most of the research projects concentrated on hardware aspects. The results of these projects, massively parallel computer systems based on various architectures, provide a large computing power, but they are not broadly accepted up to this day, especially because of the costly development process. Due to the complex runtime behaviour caused by communication overhead and load imbalance, developing an efficient parallel version of an algorithm takes a lot of time. G. Papadopoulos from MIT commented this fact with the sentence: "It appears to be easier to build parallel machines than to use them." [49]

In order to design efficient parallel programs, it is often useful to exploit different kind of potential parallelism. Depending on their different nature, i.e., especially depending on their granularity, detection mechanisms for potential parallelism are quite different varying from automatic detection to user specification. At the finest granularity level, *operation parallelism* is detected by a compiler during the code generation and used by an instruction scheduler to exploit different functional units of a processor.

Data parallelism occurs when the same operations have to be applied to different data and it can be detected by parallelizing compilers using loop transformation techniques. Data parallelism means that the same operations are executed, independently from each other, by different processors on different data. The granularity level reached depends on the amount of data which can be processed by a processing unit.

Task parallelism (often also called function, method, or control parallelism) represents parallelism on program parts' level, i.e., independent program parts can be executed on different processors or processor groups, where processors of the same group interact in a data parallel manner. Task parallelism has a coarse granularity, most applications have only a small degree of task parallelism. Nevertheless, the simultaneous exploitation of task and data parallelism can lead to significantly faster programs than the sole exploitation of data parallelism [52, 53]. This is especially interesting for parallel machines with a large number of processors.

Important classes of applications imply a potential of mixed task and data parallelism. In the area of scientific computing there are, e.g., methods for ordinary differential equations like extrapolation methods, iterated Runge-Kutta methods [56], or implicitly iterated Runge-Kutta methods [55], methods which exhibit a medium grain parallelism. These methods compute several independent approximation vectors in each time step and determine the final solution vector by combining them. The approximation vectors computed in each time step are independent and the computations can be performed concurrently. In the area of physical simulations there are several applications which combine different simulation methods. For example, environmental models [6] combine atmospheric, surface water, and ground water models, each requiring different numerical simulations which can be computed in parallel to each other.

In principle, there are two different approaches for implementing parallel applications: a bottom-up oriented technique starts with a non-parallel implementation of an application and detects, usually via loop transformation techniques, the application's potential of (essentially data) parallelism. A top-down oriented approach starts with a user provided parallel description of the application and adapts via program transformation techniques the degree of parallelism to the possibilities of a target machine. This approach exploits in essence task parallelism, but it also offers the possibility of exploiting parallelism at data level by using data parallel program parts.

In this thesis, we describe a tool for generating programs with mixed task and data parallel execution on distributed memory machines (DMMs) based on the TwoL (*Two L*evel parallelism) model proposed by Rauber and Rünger [54]. Our system is based on the concepts of exploitation of task and data parallelism, separation of design decisions and implementation, use of imperative programming language augmented by message passing library [28, 29]. The generation of parallel programs with mixed task and data parallelism starts from a user provided high level specification which is subsequently automatically translated with the herein described compiler tool into a description of a parallel implementation. The implementation description, containing task and data parallel parts, can then be translated into message passing programs based on hierarchically structured communication contexts. The separation between task and data parallel level is maintained during the entire program generation process, allowing at the same time specific information flow between the levels. The automatic translation steps are guided by runtime estimation formulas which include a concise model of the target machine.

Even though the detection and efficient exploitation of different kinds of parallelism often lead to efficient parallel programs, right from start, when generating a parallel program, it is difficult to determine how to exploit the parallelism detected, in order to obtain an optimal implementation on a specific parallel machine. Using pure data parallelism, we have small computation costs and possibly high collective communication costs. The problem of finding a data distribution that minimize the resulting communication overhead is NP-complete [43, 42]. Using task parallelism, we reduce the number of participating processors for a task and this on its part reduces the internal communication costs. On the other hand, a reduced number of processors increases the pure computation time. The management of processor groups involved, however, also causes some overhead. The problem of selecting the most efficient execution order and to determine the size of processor groups accordingly is NP-complete [26].

We propose an algorithm based on a genetic algorithm [33] approach that provides a static scheduling solution. Our algorithm not only determines an execution order of a given set of tasks and assigns processor groups for the execution of individual tasks, but also selects the suitable implementation for each particular task, and specifies the data distributions.

I thank my advisor, Thomas Rauber, for introducing me in the research area of parallelizing compilers and for providing agreeable and fruitful working conditions. I thank Gudula Rünger and Wolf Zimmermann for their support.

2 Related Work

Work related to the TwoL approach includes studies on new parallel programing paradigms [47], computation models [3, 20, 69], performance prediction techniques [24, 37, 25, 50], scheduling tools, and parallelizing compilers.

There are various problems to investigate, when constructing parallelizing compiler tools. In the attempt of offering a tool for parallel programing, the systems emphasize different aspects of the problem: high level coordination language [4, 40, 39, 14, 5], partitioning, placement, and code generation [11, 41], monitoring and debugging [10, 13, 18].

PARSA (PARallel program Scheduling and Assessment) environment [63] assists on the developing of parallel programs in distributed memory multiprocessors systems. To achieve this goal, PARSA is a visual, interactive environment to assist users in scheduling their parallel programs on a target architecture. The interactive nature of PARSA allows the choice of many scheduling methods. In addition, PARSA provides an environment that allows users to play "what if" scenarios in order to evaluate or fine-tune their parallel programs and choose a suitable architecture for the execution of the application.

One of the basic ideas of our system is to eliminate the need for system architecture and scheduling considerations during the programing phase of parallel application development. A similar basic idea can be found in PARSA. Although the PARSA approach and TwoL are similar in spirit, there are important differences. The objectives of PARSA and TwoL, to assist users in the development of their parallel programs on a target architecture, as well as the idea to allow users to specify an application without regard to system architectural considerations, are resembling. But whereas PARSA uses data parallelism and works on data parallel applications at operation level, TwoL combines task and data parallelism and works mainly at task level. The systems already start with the user specification on the respective granularity level and keep this level in all subsequent steps.

Integration of data and task parallelism has been an active research topic in the last years. In the area of parallelizing compilers there are compiler based approaches, like Parafrase-2 [48], Fx [66], and Paradigm [8], language based approaches, like Fortran M [31] and Opus [45], and graphical approaches like Opus [17], CODE [7], and CASCH [38]. In the following, we compare our approach to language and compiler based approaches which included support to combine task and data parallelism.

Parafrase-2 is a multilingual vectorizing/parallelizing compiler [48] implemented as a source to source code restructuring tool. Parafrase-2 distinguishes data (or horizontal) partitioning and task (or vertical) partitioning. These terms correspond to the TwoL terms data parallelism and task parallelism, respectively. Parallelism in Parafrase-2 is obtained via automatic parallelism detection in programs and via program transformation. This feature is an important difference to TwoL, where parallelism is user provided. Parafrase-2 has a convenient user interface and provides a way for user interaction at several levels during the transformation process. Although the partitioning (packaging of parallelism) phase is carried out at compile time, task scheduling is considered to be a dynamic process. Tasks which are ready to execute are queued in a ready-task queue. Each idle processor tries to dispatch the next available task from the queue. In contrast, TwoL provides a static scheduling process which decides to compile time for task order and for load balancing.

Fortran M [31, 32] is a language based approach basically consisting on a small set of extensions to Fortran. Fortran M provides language constructs for creating tasks and channels and for sending and receiving messages. Program components can be combined using sequential, parallel, or concurrent composition. In parallel and concurrent composition, the process serves as the basic building block. A process can encapsulate data, computation, concurrency, and communication, it has a well defined interface to the rest of the program.

In contrast to the Fortran M approach, our approach is more directed towards the combination of fine and medium grain parallelism, in TwoL different modules cannot communicate with each other, communication is located on a higher level. Besides, Fortran M does not support the derivation of parallel implementation, i.e., it provides mapping constructs to specify that a program executes in a virtual processor array and to locate a process within these processors, but the programer must decide on the execution order of processes and on the data distributions by himself.

Fx The Fx approach allows task parallelism by providing directives to partition processors into subgroups and to assign computations to different subgroups (task regions) [66]. Computations of a specific subgroup are executed in a data parallel way. The Fx compiler provides a mapping tool for grouping subroutine calls to modules and for mapping processors to modules [67]. Although the Fx approach is similar in spirit to the TwoL approach, there are some important differences: task regions in Fx cannot be nested lexically whereas TwoL allows in principle a hierarchical structure of modules. On the other hand allows Fx a dynamic nested partitioning of processors by allowing (recursive) procedure calls with internal partitioning of processors whereas TwoL requires all task coordination to be performed on the upper level of the program derivation process. Currently, recursive modules on the upper task coordination level are not allowed, although they can be included in principle. The Fx model is primarily a programing approach in which the programer has to decide on task partitioning and assignment of tasks to processor groups. Hence, Fx provides only a limited support for the derivation of parallel implementations. The TwoL model is more a specification approach in which the programer is responsible for specifying the available task parallelism, but the final decision whether available task parallelism will be exploited and how processors should be partitioned into groups is taken by the compiler. Therefore, TwoL provides a framework for the complete derivation process in which support tools can be integrated quite naturally. The mapping tool provided by Fx [67] is based on static runtime expressions that do not take into consideration the structure of subroutines; the subroutines' parameter are determined by separate runtime tests for each application. A model that is similar to the task parallelism model of Fx has been added to High Performance Fortran [35] as an approved extension.

Paradigm The Paradigm compiler [8, 50, 52] provides a framework that expresses task parallelism by a macro data-flow graph which has been derived from the hierarchical task graphs used in the Parafrase compiler [9]. Nodes in the macro data-flow graph correspond to basic parallel tasks or loop constructs, edges correspond to precedence relations that exist between tasks. Nodes and edges are weighted with processing and data transfer costs both of which depend on the number of processors used for the execution. Scheduling and allocation algorithms for macro data-flow graphs are described in [50, 51] considers the generation of array distributions between tasks. The allocation algorithm decides on the number of processors to use for each node and the scheduling algorithm decides on a scheme of execution for the allocated nodes. The goal is to select a strategy that minimizes the execution time of the macro data-flow graph.

There are two main differences between Paradigm and the TwoL approach. First, Paradigm expects as input a sequential program whereas TwoL starts with a specification that expresses the maximum degree of parallelism. This requires different derivation procedures in both approaches. Second, the runtime prediction of Paradigm is based on measured execution times of the tasks, whereas TwoL uses a runtime prediction method for the design of task programs.

Scheduling techniques used in compiler tools are quite different. Casavant [15] presents an interesting classification of process scheduling methods; this classification is also used in [64]. We take over these categories when comparing scheduling tools. According to this classification, our scheduling is a *global suboptimal static approximate* method where the solution space is searched according to a *genetic algorithm search* strategy.

Our scheduling is a static one, i.e., we assume that the task graph is known beforehand and it does not change during computation. Thus, our scheduling is not related to the vast literature on dynamic load balancing in parallel and distributed computing. We assume that the multiprocessor system is uniform and nonpreemptive; that is, the processors are identical and a processor completes the current task before executing a new one. Our schedule provides a mapping of tasks to processors, but above all it gives an execution ordering that minimizes the execution time of the entire task graph. Finally, we cannot fairly characterize our scheduling by simply comparing scheduling results with results obtained by other systems from the literature; our scheduling is a multi-dimensional scheduling, it provides task ordering, mapping, but it also selects implementation versions with appropriate data distributions. Thus, because at most partial results could be compared, we leave such comparisons and limit to analyze the results.

We will compare our scheduling with suboptimal static methods using genetic algorithms as approximating method to find an optimal solution. No one of the compared methods uses various implementation variants for tasks or various data distributions for parameters, as our scheduling does.

The schedule from Hou et al. [36] works on homogenous environments, and the schedule from Wang, Siegel et al. [71] works on heterogeneous computing environments. Both approaches use tasks with known, i.e., measured and fixed, computation costs. Due to the dimension of the problem handled, the individuals of the genetic algorithm population have to code only information about tasks and about processors, whereas individuals in our system have a third dimension to code information related to implementation variants and data distribution. Fitness evaluation has to fit with the problem definition, i.e., fitness evaluation is always adjusted for the problem and herewith system specific.

Corrêa et al. [19] propose a combined approach, where a genetic algorithm is improved with the introduction of some knowledge about the scheduling problem represented by the use of a list heuristic in the crossover and mutation genetic operations. Dhodhi et al. [22] have a similar idea, their technique is based on problem-space genetic algorithms. It combines the search power of genetic algorithms with list scheduling heuristic in order to reduce the completion time and to increase the resource utilization. These systems, also, work with fixed computation costs. Our system uses runtime prediction formulas which are parametrized with the actual processors assigned and the actual data distribution selected.

New scheduling approaches, presented by Ahmad et al. [1] and De Falco et al. [21], deal with parallelizing the scheduling algorithm itself. This is an interesting future development for our system, too, especially due to the fact that our genetic algorithm approach offers an almost natural parallelization interface.

3 System Overview

Our system clearly separates task parallelism from data parallelism. The separation between the two levels is preserved during the design and implementation phases while allowing specific information to flow between the two levels. Herewith, the model is able to combine well known results in the detection of data parallelism with the design of task parallelism of a specific application.

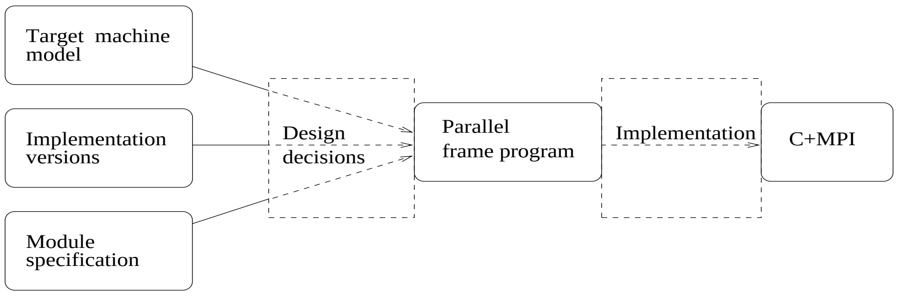


Figure 1: System overview.

This clear separation is used to divide the work between programer and compiler system. Figure 1 gives a rough overview of the system. The programer specifies an algorithm as a *module specification*, i.e., a specification of the algorithm with the maximum degree of task parallelism available. Additionally, the programer provides a specification for the data parallel modules used.

The compiler translates the module specification into a full specification of a parallel program, the *parallel frame program*, with mixed task and data parallelism. Here, several *implementation versions* for the tasks can be taken into consideration. The design decisions taken include scheduling of independent tasks, assignment of processor groups to tasks, selection of appropriate implementation versions for tasks, and data distribution specification.

The design decisions for exploiting parallelism are entirely separated from the generation (*implementation*) of parallel code. The result of the design decisions step, the parallel frame program, can be translated by a syntax directed pass in any imperative language augmented with a message passing library supporting groups. We have chosen C as the imperative language and the message passing interface MPI.

Figure 2 gives a more detailed overview of the parts of the system described in this thesis. This figure also illustrates in which order this thesis treats the different system aspects.

- The *module specification*, including the syntax of the module specification language used by the programer for the method's specification, is described in Section 4.
- The *module graph* (Section 5) is an intermediate data structure which makes module specification information easier accessible and which transfers relevant information from the module specification to the subsequent design decision step.

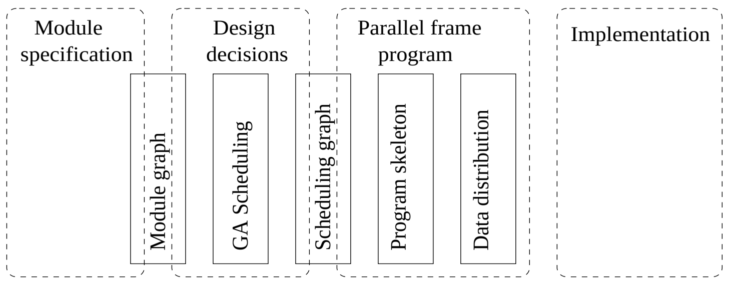
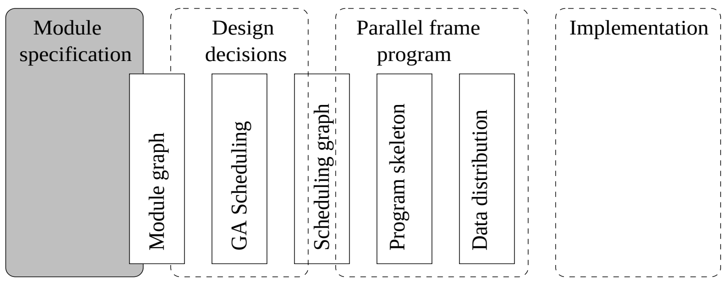


Figure 2: System elements.

- Fundamental information related to the *design decisions* step, especially definition of the scheduling problem, description of the runtime estimation, and specification of the target machine, can be found in Section 6.
- The main design decision problems, i.e., task scheduling, assignment of processor groups to tasks, selection of appropriate implementation versions, and data distribution specification, are resolved by a genetic algorithm approach (denoted in the figure as *GA scheduling*)which is described in Section 7.
- Relevant scheduling information is summarized into a *scheduling graph*. This intermediate data structure is used to concentrate design decisions, eliminate redundancy, and make design information easier accessible. See therefore Section 8.
- Design decisions are fixed in *parallel frame programs*. The syntax of the description language used is described in a separate section, see therefore Section 9.
- The algorithm which fixes the implementation information into a parallel frame program has several distinct steps described sequentially. First, we transform the scheduling graph information and construct a parallel frame *program skeleton*. This step is described in Section 10. Subsequently, *data distribution* modules are added and the therefore used communication environment is established. These steps are detailed in Section 11.
- The *implementation* of parallel frame programs into C+MPI is a simple syntax driven program transformation step, not described in this thesis.



4 Module Specification

In this section we describe the syntax of the module specification language, a small coordination language. The programer uses this language to specify on a high level the method to be implemented. The module specification indicates the maximum degree of task and data parallelism within the specified method. Task parallelism is indicated by corresponding language concepts, data parallelism is "hidden" inside data parallel program parts.

4.1 Syntax of the Module Specification Language

Module specification A *module specification* is a non-executable program which (only) indicates the maximum degree of parallelism without giving an exact execution order of tasks or specifying a data distribution of variables. A module specification (Figure 3 specifies the language's syntax) is hierarchically structured and consists of a set of composed modules.

Composed modules consist on a number of tasks where each task is either an activation of another composed module or of a basic module. Composed modules only specify the structure of the task parallelism, they do not contain direct operations on data. Tasks in composed modules may be executed concurrently ($\|$ operator, `parfor` loop) or sequentially ($\circ$ operator and `for` loops). There is no dependence between modules executed in parallel.

Basic modules Direct operations on data are hidden within basic module activations. *Basic modules* are data parallel programs, i.e., the same operations are applied to multiple elements of a data structure. Basic modules are provided as executable parallel programs parts or functions operating on arbitrary data, e.g., taken from a library, or in form of a high-level data parallel specification (cf. [29] for details).

Module expressions are expressions consisting of module activations executed concurrently or sequentially. Syntactically, module expressions are `statements` (cf. Figure 3).

```
module_specification→ program identifier comp_module_set
comp_module_set →     comp_module comp_module_set
                      | ϵ
comp_module →         module module_decl
                      assign statements end
module_decl →         identifier ( parameters )
parameters →          param param_list
                      | ϵ
param_list →          , param param_list
                      | ϵ
param →               param_type identifier : data_type
param_type →          in | out | ϵ
data_type →           struct_type elem_type
struct_type →         mat | vec | ϵ
elem_type →           char | int | float | double
c_parameters →        identifier id_list
                      | ϵ
id_list →             , identifier id_list
                      | ϵ
statements →          stmt stmt_list
stmt_list →           op_sign stmt stmt_list
                      | ϵ
op_sign →             ||  |  ○
stmt →                identifier ( c_parameters )
                      | { statements }
                      | for ( enum_expr ) statements
                      | parfor ( enum_expr ) statements
                      | if ( simple_expr ) statements
simple_expr →         identifier relop identifier
                      | ϵ
enum_expr →           identifier = bound .. bound
                      | ϵ
bound →               identifier | number
comments →            [ text without brackets ]
```

Figure 3: Syntax of the module specification language.

Restrictions The module specification does not contain any specification which anticipate implementation decisions. Module activations specify the I/O behaviour of the activated composed or basic module, without giving details on data distribution or processor groups used. Variables in conditional expressions for loops and for conditionals (`enum_expr` and `simple_expr`) are restricted to be scalars only. Additionally, these expressions do not contain variables or values which refer to particular processors or to special data distributions. Currently, recursive modules are not allowed, although they can be included in principle.

In order to keep the language simple, we use only a few elements. Not included structures like, e.g., the conditional's else-part or while-loops, may be expressed by using the language elements.

4.2 Example of a Module Specification

```
module CG (in A:mat double,p:vec double,$p_0$:vec double,w:vec double,
              $w_0$:vec double,$r_0$:vec double,x:vec double,$\lambda_0$:double,
              out p:vec double,x:vec double,r:vec double)
assign
{ for (k = 0..K)
  { mv_prod (A,p,w)
   ∘ { vv_prod (p,$r_0$,$tmp_1$)
       || vv_prod (w,p,$\lambda_1$)
       || vv_prod (w,w,$tmp_2$)
       || vv_prod (w,$w_0$,$tmp_3$)
     }
   ∘ { { op_assign ($tmp_1$,op,$\lambda_1$,$\xi$)
          ∘ { { sv_prod ($\xi$,p,$tmp_4$)
                ∘ vv_add (x,$tmp_4$,x)
              }
              || { sv_prod ($\xi$,w,$tmp_5$)
                   ∘ vv_sub (r,$tmp_5$,r)
       }   }   }
       || { { { op_assign ($tmp_2$,op,$\lambda_1$, $\mu$)
                ∘ sv_prod ($\mu$,p,$tmp_6$)
              }
              || { { op_assign ($tmp_3$,op,$\lambda_0$, $\nu$)
                     || vv_assign (w,$w_0$)
                   }
                   ∘ { sv_prod ($\nu$,$p_0$,$tmp_7$)
                       || assign($\lambda_1$,$\lambda_0$)
            }   }   }
            ∘ { vv_add ($tmp_6$,$tmp_7$,$tmp_6$)
                || vv_assign (p,$p_0$)
              }
            ∘ vv_sub (w,$tmp_6$,p)
  }  }    }
}
end
```

Figure 4: Composed module for conjugate gradient iteration.

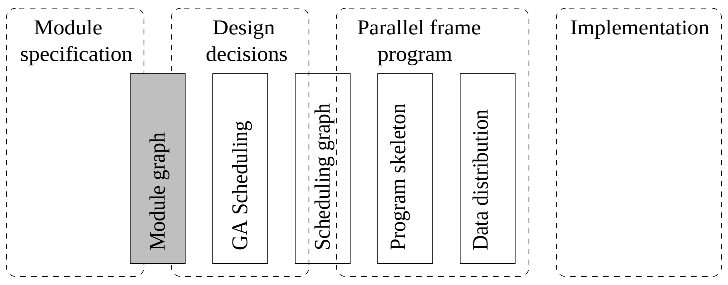
The conjugate gradient method (CG) is a Krylow-space method [70] to solve a system of linear equations $Ax = b$ with a symmetric, positive-definite coefficient matrix $A \in \mathbb{R}^{M\times M}$. In each iteration step k the method chooses a search direction $p_k \in \mathbb{R}^n$ and computes a new approximation $x_k \in \mathbb{R}^n$.

Module specification for the CG method Figure 4 shows a possible module specification of the program for the CG method. The composed module CG has an I/O behaviour which is specified by in/out parameters. There are basic modules for matrix-vector multiplication (mv_prod), for computing the scalar product (`vv_prod`), for vector addition and subtraction (`vv_add`, `vv_sub`), for multiplication of a vector with a scalar (`sv_prod`), and for assignment of values for various data structures (`vv_assign`, `assign`, and `op_assign`). In an iteration step, the `mv_prod` module (line 04) must be executed first, before the `vv_prod` modules can be started. The different instances of `vv_prod` (lines 05 to 08) may be executed in parallel. The computation continues with the execution of a block (lines 10 to 30) containing two blocks executed concurrently. These blocks (lines 10 to 16, and 17 to 30, respectively), on their part, contain several basic modules executed sequentially or concurrently.

4.3 Section's Summary and Outlook

In this section we described the syntax of the module specification language. The language concepts, especially the $\|$ and $\circ$ operators, offer the possibility for describing task and data parallelism available within the specified method. The programer provides for the maximum degree of parallelism of the method. Subsequently, in several transformation and design decision steps, the system will adapt the possible parallelism to the actual possibilities of a target machine.

As a larger example we gave the module specification of the conjugate gradient method. Subsequent examples will demonstrate the evolution of this module specification through the system.



5 Module Graph

In this section we describe a data structure which will be used by the next phases of the system and which offer the opportunity of easily accessing the information given by the user by means of the module specification.

In the following, first, we define some notions, thereafter we describe the graph construction and specify information stored by the graph. The module graph is constructed recursively, the rules to be applied to the basic language components are described in detail.

From the design decisions point of view, a composed module contains two significant information: the possible task execution order and the parameter dependence. For our purposes, we concentrate this information into a directed acyclic graph. Beside the graph information, additional data concerning the module nodes is stored in annotations attached to each one of the nodes.

Module graph The *module graph* is an intermediate data structure which transfers relevant information from the module specification to the design decision step. We construct a module graph for each composed module. For a module specification program consisting of several composed modules, several independent module graphs are constructed.

Input/output modules For a module expression `M` we recursively define, over the structure of the expression, the set `IM` of *input modules* and the set `OM` of *output modules*. Intuitively, the set of input/output modules consists of the modules which are "visible" from outside the module expression. The formal definition is as follows ($\mathtt{M_1}$ and $\mathtt{M_2}$ are module expressions, too):

$$\mathtt{IM(M)} = \begin{cases} \mathtt{A} & \text{for } M = A, \text{ A basic module} \\ \mathtt{IM(M_1)} & \text{for } M = \{M_1\} \\ \mathtt{IM(M_1)} \cup \mathtt{IM(M_2)} & \text{for } M = M_1 \parallel M_2 \\ \mathtt{IM(M_1)} & \text{for } M = M_1 \circ M_2 \end{cases}$$

$$\mathtt{OM(M)} = \begin{cases} \mathtt{A} & \text{for } M = A, \text{ A basic module} \\ \mathtt{OM(M_1)} & \text{for } M = \{M_1\} \\ \mathtt{OM(M_1)} \cup \mathtt{OM(M_2)} & \text{for } M = M_1 \parallel M_2 \\ \mathtt{OM(M_2)} & \text{for } M = M_1 \circ M_2 \end{cases}$$

5.1 The Nodes

Nodes In principle, *nodes* of the module graph correspond to module activations inside the module expression. In the following we do not distinguish between a node of the module graph and the corresponding module activation of the composed module expression as long as the assignment is unambiguous.

The module graph contains one node for each module activation occurring in the module specification program. For other language components, e.g., composed modules and loops, there are special rules for the generation of the corresponding nodes. These rules are described in Section 5.3 below.

Parameter lists For a node `A`, `IN(A)/OUT(A)` represents the *input/output parameter list* of the node. Let `a` be the i-th parameter of `A`. We define (cf. Figure 5 for an example):

$$\texttt{IN(A)[i]} = \begin{cases} \texttt{a} & \text{a is an input parameter} \\ \texttt{NIL} & \text{else} \end{cases}$$

$$\texttt{OUT(A)[i]} = \begin{cases} \texttt{a} & \text{a is an output parameter} \\ \texttt{NIL} & \text{else} \end{cases}$$

module declaration: `mv_prod (`**in** `A:mat double,p:vec double,`**out** `w:vec double)`

```
IN(mv_prod)[0] = A        OUT(mv_prod)[0] = NIL
IN(mv_prod)[1] = p        OUT(mv_prod)[1] = NIL
IN(mv_prod)[2] = NIL      OUT(mv_prod)[2] = w
```

Figure 5: Example of an input/output parameter list.

Annotations The nodes of the module graph store in *parameter annotations* additional information about the I/O parameters of the corresponding module activations. The data type description of the parameters, especially the dimension of variables, is taken from the parameter description of the corresponding module activation.

5.2 The Edges

The module graph contains two kind of edges: structure edges and data edges. The computation order, specified by the structure and the operators of the composed module, is expressed by so-called structure edges, the parameter dependence is expressed by data edges.

Structure edges , drawn in the figures in this section by solid lines, result from the structure of the module expression. Structure edges illustrate precedence relations between modules, expressing whether a module has to be executed before another one.

Modules connected via a $\|$ operator do not have any connecting structure edges. For a module expression `A∘B`, with a sequential compose operator, `A`, `B` basic modules, the module graph contains the oriented structure edge `(A,B)`. In general, for a module expression $\mathtt{M}_1\circ\ \mathtt{M}_2$, the graph contains the set of oriented structure edges $\{(\mathtt{A}_1,\mathtt{A}_2);\ \mathtt{A}_1 \in \mathtt{OM}(\mathtt{M}_1),\ \mathtt{A}_2 \in \mathtt{IM}(\mathtt{M}_2)\ \}$, see as an example Figure 6.

Corresponding
module specification

$\{\,\mathtt{A}_1\ \|\ \mathtt{A}_2\ \|\ \mathtt{A}_3\,\}$
$\circ\,\{\,\mathtt{B}_1\ \|\ \mathtt{B}_2\,\}$

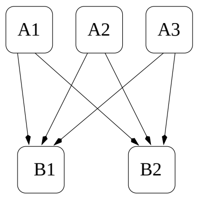


Figure 6: Module graph (structure edges).

Data edges, drawn in the figures by dashed lines, describe the parameter data dependence. We use the usual visibility rules for variables in nested blocks, the notions of *successor* and *predecessor* have to be applied accordingly. Note that the successor/predecessor relation between module activations is given by the structure edges between the corresponding nodes. If a module activation `A(...,[out]a)` with an output parameter `a` is predecessor of module activation `B([in]a,...)` with an input parameter `a` and there is no module activation `C` with the following property: `C` is successor of `A`, `C` is predecessor of `B`, and `C` has an output parameter `a`, then the directed data edge `(A,B)` has to be added to the module graph.

Annotations to data edges store information about the corresponding parameters (cf. Figure 7); in the sections below we give additional information about the data stored.

Corresponding module specification
[for a better readability:
in/outs are indicated]

{ `A([in] a, b, [out] c)` $\|$ `B([in] a, [out] d)` }
∘{ `C([in] d, out ...)` $\|$ `D([in] c, d, [out] ...)` }

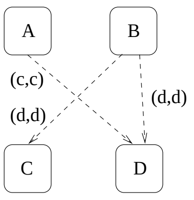


Figure 7: Module graph (data edges).

Data operations A data edge `(A, B)` between two modules represents a data transfer from module `A` to module `B`. Usually, such a transfer involves data distribution operations. We denote with *data operations of an edge* `(A, B)` data transfer operations from module `A` to module `B` represented by the edge `(A, B)`, i.e., data distribution operations concerning the parameters mentioned in the annotations of `(A, B)`.

5.3 Composed Modules, Loops, and Conditionals

5.3.1 Composed Modules

Input/output node A composed module C is represented by two modules:

IM(C) = C_0, OM(C) = C_1.

The corresponding nodes C_0, C_1 represent the *input/output node* of the graph, respectively. These two nodes represent the "hull" of the module graph, they are the connection points for the graph to the environment; edges, i.e., structure and data edges, to/from the composed module C end/start at the node C_0/C_1, respectively.

Let M be the module expression inside the composed module. There are structure edges from C_0 to M and from M to C_1, namely the structure edges $\{(C_0,A);\ A \in IM(M)\}$ and $\{(A,C_1);\ A \in OM(M)\}$. Example: in Figure 8 the structure (solid lines) edges (C_0,A) and (A,C_1).

Parameters There are also special rules to be applied to the parameters of the composed module C. We define:

IN(C_0) = IN(C), OUT(C_0) = NIL,
IN(C_1) = NIL, OUT(C_1) = OUT(C).

The descriptions for the parameters in IN(C_0) and OUT(C_1) take the description of the corresponding parameters of C.

If a is an input parameter of C and A is a module activation inside the composed module having a as an input parameter, and there is no module activation B with: B is successor of C_0, B is predecessor of A, and B has an output parameter a, then an oriented data edge (C_0,A) is generated. Analogously, for an output parameter of the composed module an oriented data edge (A,C_1) is generated.

Note that in the case of the composed module, we connect *input* parameters of the composed module to *input* parameters inside the composed module and *output* parameters to *output* parameters.

The corresponding module specification
[for a better readability: in/outs are
indicated, data types are left out]

```
module C (in a, out c)
assign
{ A([in] a, [out] b)
  ∘ B([in] a, b, [out] c)
}
end
```

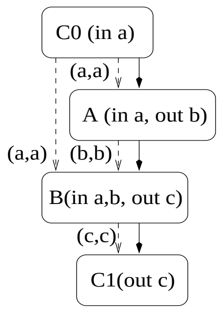


Figure 8: Module graph for a composed module.

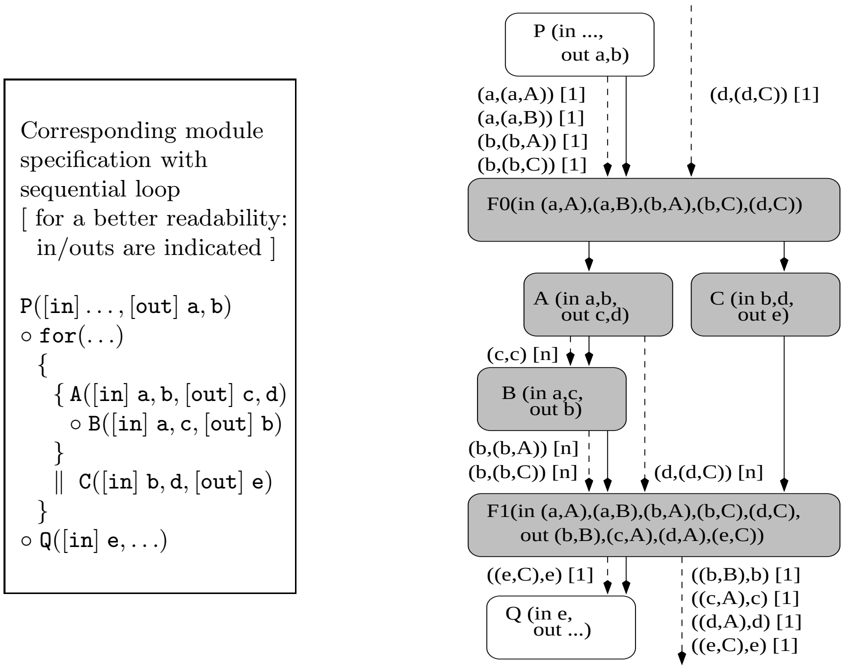
Example In Figure 8 the data (dashed lines) edges (C_0,A) and (C_0,B) for the input parameter a, and data edge (B,C_1) for the output parameter c are inserted.

5.3.2 Sequential Loops

The idea behind the construction of subgraphs for sequential loops is to offer a possibility to specify data distribution operations caused by looping, without using backward edges and, thus, preserving an acyclic graph. Therefore, we splitt the loop in two modules representing the beginning and the end of the loop, respectively. Data distribution operations caused by the looping are substitute by data operations from the inside of the loop towards the loops' end module.

Input/output nodes A sequential *for loop* `F` is represented by two modules $\mathtt{F}_0$ and $\mathtt{F}_1$; module $\mathtt{F}_0$ represents the *input module*, module $\mathtt{F}_1$ represents the *output module*, i.e.,

$$\mathtt{IM(F)} = \mathtt{F}_0, \qquad \mathtt{OM(F)} = \mathtt{F}_1.$$



```
Corresponding module
specification with
sequential loop
[ for a better readability:
  in/outs are indicated ]

P([in]..., [out] a,b)
∘ for(...)
  {
   { A([in] a,b, [out] c,d)
     ∘ B([in] a,c, [out] b)
   }
   || C([in] b,d, [out] e)
  }
∘ Q([in] e,...)
```

Figure 9: Module graph containing the subgraph for a sequential loop.

Figure 9 illustrates a subgraph corresponding to a fragment of a module specification containing a sequential loop. The nodes corresponding to $\mathtt{F}_0$ and $\mathtt{F}_1$ represent the interface of the sequential loop to module activations of the composed module, i.e., they are the connection points for the subgraph. Structure edges from other modules of the composed module directed to the for loop ends at the node $\mathtt{F}_0$. Structure edges from the for loop directed to other modules of the composed module begins at the node $\mathtt{F}_1$. We construct the subgraph for the sequential loop as if the expression $\mathtt{F}_0 \circ \mathtt{M} \circ \mathtt{F}_1$ would be part of the composed module (`M` denotes the module expression inside the loop).

Parameters $\mathtt{F}_0$ and $\mathtt{F}_1$ represent the *loop's interface*; we determine the I/O parameters of the loop and assign them to the new nodes. Data edges from outside the loop end at $\mathtt{F}_0$, data edges from the loop towards outside the loop begin at $\mathtt{F}_1$.

The *input parameters* of the loop are input parameters of module activations from inside the loop. These parameters get in the first iteration step their values from outside the loop. In subsequent iteration steps, they (partly) may get the values from inside the loop. These parameters are the input parameters of node $\mathtt{F}_0$ and $\mathtt{F}_1$. For example, in Figure 9 the input parameters of the loop include all input parameters of module `A`, all input parameters of module `C`, and from the input parameters of module `B` only those parameters which are not "fed" by output parameters of module `A`, i.e., from module `B` only parameter `a` (parameter `c` gets its value from the output parameter `c` of module A).

The *output parameters* of the loop are output parameters of the module activations from inside the loop. Those are the output parameters of node $\mathtt{F}_1$. In Figure 9, the list of output parameters includes the output parameters of activations of modules `A`, `B` and `C`.

We are interested in the set of *parameters* "visible" outside the loop, i.e., input parameters which get their values from outside the loop and output parameters which give their values to module parameters outside the loop. The idea is that in order to represent the interface from the modules inside the loop to the modules outside the loop, $\mathtt{F}_0$ and $\mathtt{F}_1$ need a precise description of the parameters from inside the loop. We need for these parameters not only the name but also additional information like, e.g., the data distribution. This information is bound to the corresponding module activation.

In order to obtain this information, we enlarge for these nodes the information stored in parameter lists and we define $\mathtt{IN}_\mathtt{p}/\mathtt{OUT}_\mathtt{p}$ as being the *precise input/output parameter list* which considers not only the parameter name but also the corresponding parameter description taken from the corresponding module activation. Thus, an element `(a,A)` of a precise list includes not only `a`, the parameters name, but also `A`, the id of the node corresponding to the concerned module activation. See, e.g., in Figure 9 the parameter list for the node $\mathtt{F}_1$: `(a,A)` means the parameter `a` with type and data distribution specified at the activation of module `A` (in this example we do not make a difference between module activation and the corresponding node id, unless the correspondence is not clear), `(a,B)` means the parameter `a` with type and data distribution specified at module `B`.

Let `M` be the module expression inside the loop and let $\mathtt{IN}_\mathtt{p}(\mathtt{M})/\mathtt{OUT}_\mathtt{p}(\mathtt{M})$ be the precise input/output parameter list of the loop. According to the explanations above, we have:

$$\begin{aligned} &\mathtt{IN(F_0)\ =\ IN_p(M)}, &\qquad& \mathtt{OUT(F_0)\ =\ NIL}, \\ &\mathtt{IN(F_1)\ =\ IN_p(M)}, &\qquad& \mathtt{OUT(F_1)\ =\ OUT_p(M)}, \end{aligned} \qquad (1)$$

i.e., the precise input parameter list of the loop is assigned as input parameter list to both nodes $\mathtt{F}_0$ and $\mathtt{F}_1$, the precise output parameter list of the loop is assigned as output parameter list to output node $\mathtt{F}_1$. Input node $\mathtt{F}_0$ has an empty output parameter list.

Construction of precise parameter lists As next, we describe in detail the construction rules for obtaining the precise lists $\mathtt{IN}_\mathtt{p}\mathtt{(M)}$ and $\mathtt{OUT}_\mathtt{p}\mathtt{(M)}$. In order to construct a precise list, we first construct the set of non NIL elements included in the list and subsequently we generate the precise list by sorting the elements and adding the NIL elements required by the lists definition. We denote with $\texttt{Set}(\texttt{L}_\texttt{p})$ the *set of non NIL elements* of a precise list $\texttt{L}_\texttt{p}$.

First, we introduce the operation $\setminus_c$, a *coarse minus.* When applied to two sets $\texttt{A}_\texttt{p}$, $\texttt{B}_\texttt{p}$ of precise elements, the operation eliminates the elements having correspondings with the same parameter name in the second set. The operation is defined as follows:

$\texttt{A}_\texttt{p} \setminus_c \texttt{B}_\texttt{p} = \{ \texttt{(a,A)} \in \texttt{A}_\texttt{p};$ there is no element $\texttt{(a,B)} \in \texttt{B}_\texttt{p} \}$.

For example, let be the following fragment of a module specification (A, B, C, and X are module activations, in/out parameters are indicated in comments):

$\texttt{X([out]a, c)} \circ \{\texttt{A([in]a, b)} \parallel \texttt{B([in]a)} \parallel \texttt{C([in]b)}\}$.

By using the operation $\setminus_c$ we determine the set of parameters which gets their input values from modules located before module X:

$\{\texttt{(a,A)},\texttt{(a,B)},\texttt{(b,A)},\texttt{(b,C)}\} \setminus_c \{\texttt{(a,X)},\texttt{(c,X)}\} = \{\texttt{(b,A)},\texttt{(b,C)}\}$.

The construction of sets of precise elements is recursive, following the structure of the module expression concerned. For M, $\texttt{M}_1$, $\texttt{M}_2$ module expressions, A module activation, we define recursively over the structure of M the sets $\texttt{Set}(\texttt{IN}_\texttt{p}(\texttt{M}))$ and $\texttt{Set}(\texttt{OUT}_\texttt{p}(\texttt{M}))$ which represent the sets of input/output parameters visible outside the concerned module expression M:

$$\texttt{Set}(\texttt{IN}_\texttt{p}(\texttt{M})) = \begin{cases} \{\texttt{(a,A); a input parameter of A}\} & \texttt{for M = A} \\ \texttt{Set}(\texttt{IN}_\texttt{p}(\texttt{M}_1)) & \texttt{for M = } \{\texttt{M}_1\} \\ \texttt{Set}(\texttt{IN}_\texttt{p}(\texttt{M}_1)) \cup \texttt{Set}(\texttt{IN}_\texttt{p}(\texttt{M}_2)) & \texttt{for M = M}_1 \parallel \texttt{M}_2 \\ \texttt{Set}(\texttt{IN}_\texttt{p}(\texttt{M}_1)) \cup (\texttt{Set}(\texttt{IN}_\texttt{p}(\texttt{M}_2)) \setminus_c \texttt{Set}(\texttt{OUT}_\texttt{p}(\texttt{M}_1))) & \\ & \texttt{for M = M}_1 \circ \texttt{M}_2 \end{cases}$$

$$\texttt{Set}(\texttt{OUT}_\texttt{p}(\texttt{M})) = \begin{cases} \{\texttt{(a,A); a output parameter of A}\} & \texttt{for M = A} \\ \texttt{Set}(\texttt{OUT}_\texttt{p}(\texttt{M}_1)) & \texttt{for M = } \{\texttt{M}_1\} \\ \texttt{Set}(\texttt{OUT}_\texttt{p}(\texttt{M}_1)) \cup \texttt{Set}(\texttt{OUT}_\texttt{p}(\texttt{M}_2)) & \texttt{for M = M}_1 \parallel \texttt{M}_2 \\ (\texttt{Set}(\texttt{OUT}_\texttt{p}(\texttt{M}_1)) \setminus_c \texttt{Set}(\texttt{OUT}_\texttt{p}(\texttt{M}_2))) \cup \texttt{Set}(\texttt{OUT}_\texttt{p}(\texttt{M}_2)) & \\ & \texttt{for M = M}_1 \circ \texttt{M}_2 \end{cases}$$

For $\texttt{F}_0$ and $\texttt{F}_1$, the input and output node of the sequential loop, we compute the *sets* $\texttt{Set(IN(F}_0\texttt{))}$, $\texttt{Set(IN(F}_1\texttt{))}$, and $\texttt{Set(OUT(F}_1\texttt{))}$ by using these construction rules and definition (1). The module expression M used in this case is the module expression inside the sequential loop. We obtain the parameter *lists* $\texttt{IN(F}_0\texttt{)}$, $\texttt{IN(F}_1\texttt{)}$, and $\texttt{OUT(F}_1\texttt{)}$ by lexicographically sorting the corresponding set elements (first sorting criterion is the parameter name and second criterion is the module name). Finally, in order to determine the parameter lists for $\texttt{F}_0$ and $\texttt{F}_1$, we have to determine the order of the elements. Therefore, the input parameters are listed first, followed by the list of output parameters. As an example, in Figure 9 the parameter lists of the nodes $\texttt{F}_0$ and $\texttt{F}_1$ are specified in detail.

Nodes inside the sequential loop Nodes representing modules inside the sequential loop are generated as usual. Their only special property is that modules inside the loop are performed repeatedly. A *repeating annotation*, attached to each one of the nodes inside the sequential loop, stores the number of loop repetitions. If this information is not explicitly mentioned in the loop specification, then a default value is assumed.

Structure edges Let M be the module expression inside the loop. Structure edges inside M are constructed as usual. Additionally, there are structure edges for connecting the nodes F_0 and F_1 to the nodes of M. They are constructed as if the expression $F_0 \circ M \circ F_1$ would be part of the composed module, i.e., we construct the structure edges $\{(F_0,A);\ A \in IM(M)\}$ and $\{(B,F_1);\ B \in OM(M)\}$. Example: in Figure 9 these are the edges (F_0,A), (F_0,B), and (B,F_1), (C,F_1), respectively.

Data edges The repeated operations of a sequential loop are 'emulated' by the module graph in two ways: by nodes with a repeating annotation and which represent the modules inside the loop, and by data edges which represent the data distribution operations. There are several kinds of data edges, depending on the data distribution operations represented: distributions performed only once at the beginning or at the end of the loop, and distributions performed in each loop iteration. In the following we describe each data edge type in detail. Let M be the module expression inside the loop. All examples given below refer to Figure 9. In this figure, we specified in brackets attached to each data edge, the number of iterations to be performed for the data operation represented: [n] represents a data operation which has to be performed for each loop iteration, [1] means a data operations which is performed only once. Actually, this information is derived from the repeating annotation (see above) attached to nodes.

- There are data edges inside the subgraph corresponding to M, i.e., oriented data edges (A,B) with A and B nodes corresponding to module activations in M. These edges are generated as usual. Data operations represented by these edges have to be performed for each loop iteration.
  Example: data edge (A,B) for parameter c.

- There are data edges which represent data operations caused by connecting the modules inside M to modules outside the loop. The corresponding data operations are performed only once, at the beginning or at the end of the loop.

  Data edges for input data for module activations inside M, when data is coming from module activations outside the loop, are represented by oriented edges $\{(A,F_0);$ A a node outside the loop subgraph$\}$. The input parameters of node F_0 take the description of the corresponding input parameters in M. This feature is preserved in all subsequent processes, cf. Section 5.3.3. Data operations represented by these edges are performed only once at the beginning of the loop.

  Example: data edge (P,F_0) for parameter a of module A (annotation (a,(a,A))) and data edge (P,F_0) for parameter a of module B (annotation (a,(a,B))). Input parameter (a,A)/(a,B) of F_0 takes the description of the input parameter a of module activation A/B, respectively.

  Data edges for output data from module activations in M going to module activations outside the loop are represented by oriented edges $\{(F_1,B);$ B a node outside the loop subgraph}. The output parameters of node F_1 take the description of the corresponding output parameters in M and preserve this feature in all subsequent processes, see Section 5.3.3. Data operations represented by these edges are performed only once at the end of the loop.

Example: data edge (F_1,Q) for parameter e of module C (annotation ((e,C),e)). Output parameter (e,C) of F_1 takes the description of output parameter e of module activation C.

- There are data edges for looping, i.e., data edges which represent the data operations caused by the sequential looping, and represented by oriented edges $\{(A,F_1);\ A \in M\}$. Data operations represented by these edges have to be performed for each loop iteration. The input parameters of node F_1 take the description of the corresponding input parameters from inside M and preserve this feature in all subsequent processes.

  Example: data edge (B,F_1) for input parameter b of module A (annotation (b,(b,A))), data edge (B,F_1) for parameter b of module C (annotation (b,(b,C))), and data edge (A,F_1) for parameter d of module C (annotation (d,(d,C))).

5.3.3 Context Dependent Information

An important function of input and output nodes of a sequential loop is to transfer information about parameters inside the loop to modules outside the loop. For technical reasons, we record the information about parameters in the parameter lists of the respective input/output nodes, although the information is not local to these nodes and cannot be fixed in these nodes. All information, e.g., concerning data distribution, is fixed inside the loop at the nodes where the parameter actually belongs.

We introduce an additional annotation called *context dependence*. Nodes having this annotation have to get their information, e.g., information about data distribution of parameters or about the processor group used, from another node.

F_0/F_1, the input/output nodes of a sequential loop, are context dependent nodes. Example: in Figure 9, input parameter (a,A) of F_1 is only a wildcard, the actual parameter is input parameter a in module A. Actually, node F_1 has no information concerning a, e.g., on data distribution, so that F_1 always gets this information from A.

Besides the sequential loop input and output nodes, we will see in Section 7 that input and output nodes of a composed module are also context dependent: the input node depends from the output node and vice versa, this dependence relation is reciprocal.

5.3.4 Parallel Loops

The meaning of a `parfor` loop is that several identical module expressions can be performed concurrently. All these expressions are identical to the module expression inside the `parfor` loop. The number of existing expressions is known at compile time, i.e., as mentioned in Section 4.1, the conditional expression of the parallel loop contains only scalars and has no reference to data distribution or to processors used.

The construction of the module graph for a parallel loop is in several points similar to the construction of the graph for the sequential loop, so that we refer to the previous Section 5.3.2 and we point out only differences. These differences are mainly caused by the fact that in parallel loops there is no iteration: instances are performed concurrently, there is no data transfer between them.

A parallel *parfor loop* $\mathtt{P}$ is represented by two nodes: node $\mathtt{P_0}$ represents the *input node* of the subgraph, node $\mathtt{P_1}$ represents the *output node* of the subgraph. Let $\mathtt{M}$ be the module expression inside the loop and let $\mathtt{IN_p(M)/OUT_p(M)}$ be the precise input/output parameter list of the loop. We define:

$$\mathtt{IN(P_0)} = \mathtt{IN_p(M)}, \qquad \mathtt{OUT(P_0)} = \mathtt{NIL},$$
$$\mathtt{IN(P_1)} = \mathtt{NIL}, \qquad \mathtt{OUT(P_1)} = \mathtt{OUT_p(M)},$$

i.e., the loop's precise input parameter list is assigned as input parameter list to node $\mathtt{P_0}$, the loop's precise output parameter list is assigned as output parameter list to output node $\mathtt{P_1}$. Input node $\mathtt{P_0}$ has an empty output parameter list, output node $\mathtt{P_1}$ has an empty input parameter list.

Nodes $\mathtt{P_0}$ and $\mathtt{P_1}$ are the connection points of the parallel loop's subgraph with subgraphs corresponding to module activations of the composed module situated outside the loop. Structure edges from outside the loop ends/begins at node $\mathtt{P_0/P_1}$, respectively, i.e.,

$$\mathtt{IM(P)\ =\ P_0}, \qquad \mathtt{OM(P)\ =\ P_1}.$$

Additionally, data edges for connecting the parallel loop to subgraphs corresponding to module activations outside the loop ends/begins at $\mathtt{P_0/P_1}$, respectively. Data edges inside the loop are constructed as usual.

Corresponding module specification with parallel loop
[for a better readability: in/outs are indicated]

```
P([in] ..., [out] a, b)
∘ parfor(...)
  {
    { A([in] a, b, [out] c, d)
      ∘ B([in] a, c, [out] b)
    }
    ∥ C([in] b, d, [out] e)
  }
∘ Q([in] e, [out] ...)
```

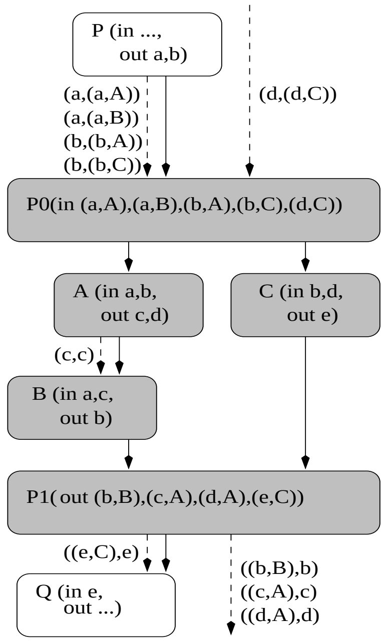


Figure 10: Module graph containing a subgraph for a parallel loop.

By using Figure 10, now we point out the specific attributes of the graph:

- There are data edges inside the subgraph corresponding to `M`, i.e., oriented data edges `(A,B)` with `A` and `B` nodes corresponding to module activations in `M`. These edges are generated as usual.
  Example: data edge `(A,B)` for parameter `c`.

- There are data edges which represent data operations caused by connecting modules inside `M` to modules outside the loop. The corresponding data operations are performed only once, at beginning or end of the loop. Note that such a data operation represents several data operations performed concurrently, the number of concurrent repetitions depends on the number of loop iterations. For output data, it is on the programer to ensure a correct data gathering.
  Example: data edge $(\mathtt{P},\mathtt{P}_0)$ for parameter `a` of module `A` (annotation `(a,(a,A))`) and data edge $(\mathtt{P},\mathtt{P}_0)$ for parameter `a` of module `B` (annotation `(a,(a,B))`). Input parameter `(a,A)`/`(a,B)` of $\mathtt{P}_0$ takes the description of the input parameter `a` of module activation `A`/`B`, respectively.
  Data edge $(\mathtt{P}_1,\mathtt{Q})$ for parameter `e` of module `C` (annotation `((e,C),e)`). Output parameter `(e,C)` of $\mathtt{P}_1$ takes the description of output parameter `e` of module activation `C`.

5.3.5 Conditionals

```
Corresponding module
specification with a
conditional
[ for a better readability:
  in/outs are indicated ]

P([in] . . . , [out] a, b)
○ if(. . .)
  {
    { A([in] a, b, [out] c, d)
      ○ B(in a, c, [out] b)
    }
    || C([in] b, d, [out] e)
  }
○ Q([in] b, c, e, [out] . . .)
```

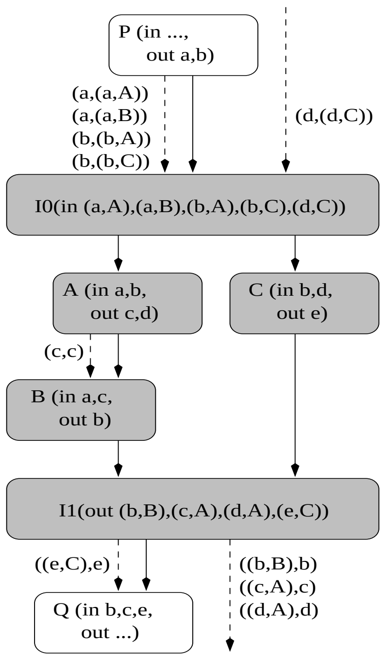


Figure 11: Module graph containing a subgraph for a conditional.

The construction of the module graph for a conditional, is similar to the constructions described in previous sections. Figure 11 illustrates a module graph containing a subgraph corresponding to a conditional. Let M be the module expression inside the conditional. The subgraph corresponding to M is constructed as usual. The frame of the subgraph for a conditional is given by two nodes: *input node* node $\mathtt{I_0}$ and *output node* $\mathtt{I_1}$ with

$$\mathtt{IN(I_0)} = \mathtt{IN_p(M)}, \qquad \mathtt{OUT(I_0)} = \mathtt{NIL},$$
$$\mathtt{IN(I_1)} = \mathtt{NIL}, \qquad \mathtt{OUT(I_1)} = \mathtt{OUT_p(M)}.$$

5.4 Example of a Module Graph

Figures 12 and 13 illustrate the module graph corresponding to the module specification of the conjugate gradient iteration specified in Figure 4. The information contained in the composed module specification is transformed into a module graph with 23 nodes (2 nodes for the composed module, 2 nodes for the sequential loop, and 19 nodes for basic module activations), 43 structure edges, and 53 data edges. In order to have a better overview of the graph, we illustrate structure and data edges in two separate figures and we leave out annotations to data edges.

5.5 Section's Summary and Outlook

In this section we described the construction of the module graph, a directed acyclic graph which records information given by the programer in the module specification. The module graph does not contain more information than the module specification but the information is easier accessible. In order to do so, some technical construction was performed. In subsequent steps we will use exclusively the module graph and we will reduce all questions concerning the initial module specification to questions solely concerning the module graph.

The construction of the module graph is a recursive process, we described in this section the construction steps for basic language elements, i.e., for modules (Section 5.1), composed modules (Section 5.3.1), loops (Section 5.3.2 and 5.3.4), and conditionals (Section 5.3.5), and we detailed the rules for connecting subgraphs together (Section 5.2).

In Section 5.4 we illustrated the module graph constructed for the conjugate gradient method.

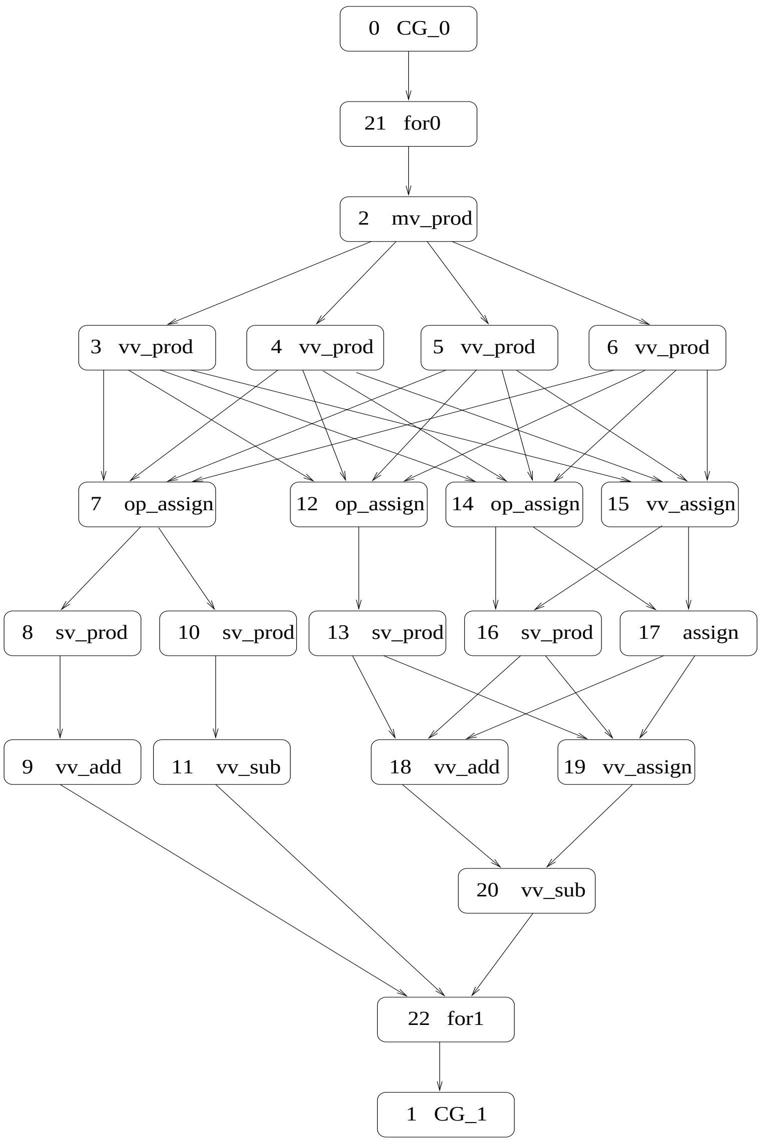


Figure 12: Module graph (only structure edges) for the conjugate gradient iteration. For the module specification confer Figure 4.

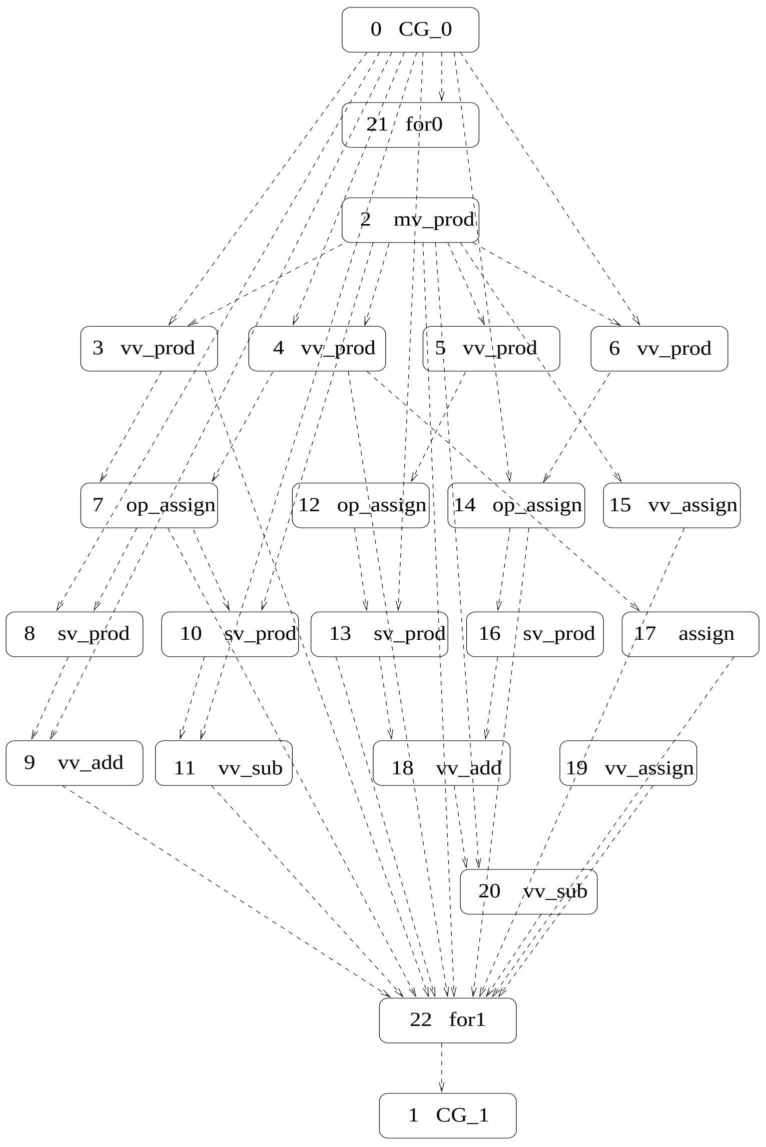
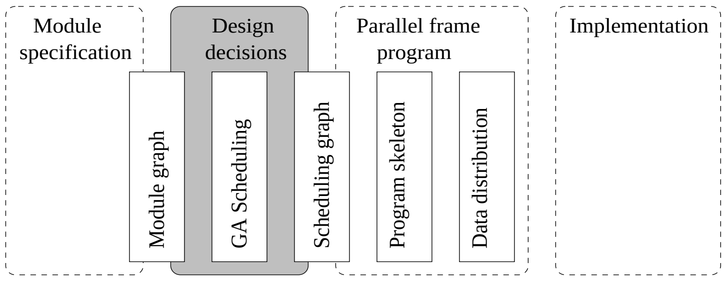


Figure 13: Module graph (only data edges) for the conjugate gradient iteration. For the module specification confer Figure 4.



6 Design Decisions

Our design decision problem consists in determining the execution order of tasks, on assigning groups of processors to tasks, and on selecting implementations from several implementation versions which differ in the number of processors and on the data distribution used, such that on a given target machine a minimal global execution time results. Because of non-linear execution times for tasks and communication operations, it is usually difficult to select the most efficient execution order and to determine the size of processor groups accordingly. Already the 'simple' problem of determining the execution order of tasks with linear execution times is NP-complete [26]. Likewise, the problem of finding a data distribution that provides a minimal communication overhead is NP-complete [43, 42].

In this section we define the scheduling problem which we want to solve and the technical frame in which design decisions are taken, i.e., we describe the runtime estimation functions used for estimating the efficienty of a solution proposed. Runtime includes computation and communication costs. Computation costs are estimated by means of runtime functions described and used in [55, 56]. Communication costs are evaluated in dependence of the amount of data transferred and of the point to point communication functions used [57, 60, 34].

Design decisions are taken for a given target machine, the machine's parameters are parameters of the runtime estimation and of the communication cost estimation function.

6.1 Scheduling Problem

Scheduling Let $\{\mathtt{t}_1, \ldots, \mathtt{t}_n\}$ be a set of tasks corresponding to module activations in a module specification, and let $\{\mathtt{v}_1^{\mathtt{t}_i}, \ldots, \mathtt{v}_{\mathtt{m}_{\mathtt{t}_i}}^{\mathtt{t}_i}\}$ be the set of implementation versions available for a task $\mathtt{t}_i$, $\mathtt{i} = 1, \ldots, \mathtt{n}$. Implementation versions are provided, for example, as library functions. In a module specification several module activations may correspond to the same module. Thus, for each one of these activations the same implementation versions are available, i.e., $\{\mathtt{v}_1^{\mathtt{t}_i}, \ldots, \mathtt{v}_{\mathtt{m}_{\mathtt{t}_i}}^{\mathtt{t}_i}\} = \{\mathtt{v}_1^{\mathtt{t}_j}, \ldots, \mathtt{v}_{\mathtt{m}_{\mathtt{t}_j}}^{\mathtt{t}_j}\}$ for $\mathtt{t}_i$, $\mathtt{t}_j$, $\mathtt{i} \neq \mathtt{j}$, being module activations corresponding to the same module.

Let $\mathtt{P}$ be a set of identical processors. The *scheduling* is a function

$$\mathtt{s} : \{\mathtt{t}_1, \ldots, \mathtt{t}_n\} \rightarrow \mathrm{I\!R} \times 2^{\mathtt{P}} \times \cup_{\mathtt{i}=1}^{\mathtt{n}} \{\mathtt{v}_1^{\mathtt{t}_i}, \ldots, \mathtt{v}_{\mathtt{m}_{\mathtt{t}_i}}^{\mathtt{t}_i}\}$$

with $\mathtt{s}(\mathtt{t_i}) = (\mathtt{s_i}, \mathtt{p_i}, \mathtt{v_i})$, $\mathtt{s_i}$ being the start time for the execution of task $\mathtt{t_i}$, $\mathtt{p_i} \subseteq \mathtt{P}$ the group of processors executing $\mathtt{t_i}$, and $\mathtt{v_i}$ an implementation version available for $\mathtt{t_i}$.

Valid schedule Let $\mathtt{comp\_time}(\mathtt{t_i}, \mathtt{v_i})$ be the *computation costs* (the computation time) which accrue when executing implementation version $\mathtt{v_i}$ of task $\mathtt{t_i}$; $\mathtt{comm\_time}(\mathtt{t_i}, \mathtt{v_i})$ denotes the *communication costs* (cf. Section 6.3) associated to $(\mathtt{t_i}, \mathtt{v_i})$.

A schedule is *valid* with reference to a given module graph if for all tasks $\mathtt{t_i},\ \mathtt{t_j},\ \mathtt{i}, \mathtt{j} = \mathtt{1}, \ldots, \mathtt{n}$, the following conditions hold (condition 1 expresses that dependent tasks have to be executed sequentially according to the precedence relations; condition 2 expresses that independent tasks must be executed by disjoint groups of processors if their execution time intervals overlap):

1. if there is an edge $(\mathtt{t_i}, \mathtt{t_j})$ in the module graph then $\mathtt{s_i} + \mathtt{comp\_time}(\mathtt{t_i}, \mathtt{v_i}) + \mathtt{comm\_time}(\mathtt{t_i}, \mathtt{v_i}) \leq \mathtt{s_j}$.
2. if there is no path $(\mathtt{t_i}, \ldots, \mathtt{t_j})$ or $(\mathtt{t_j}, \ldots, \mathtt{t_i})$ in the module graph and $\mathtt{s_i} \leq \mathtt{s_j} \leq \mathtt{s_i} + \mathtt{comp\_time}(\mathtt{t_i}, \mathtt{v_i}) + \mathtt{comm\_time}(\mathtt{t_i}, \mathtt{v_i})$ or $\mathtt{s_j} \leq \mathtt{s_i} \leq \mathtt{s_j} + \mathtt{comp\_time}(\mathtt{t_j}, \mathtt{v_j}) + \mathtt{comm\_time}(\mathtt{t_j}, \mathtt{v_j})$ then $\mathtt{p_i} \cap \mathtt{p_j} = \emptyset$.

Scheduling problem The *total execution time* of a valid schedule $\mathtt{s}$ is:

$$\mathtt{T}(\mathtt{s}) = \mathtt{max}_{\mathtt{i}=\mathtt{1},\ldots,\mathtt{n}}\{\mathtt{s_i} + \mathtt{comp\_time}(\mathtt{t_i}, \mathtt{v_i}) + \mathtt{comm\_time}(\mathtt{t_i}, \mathtt{v_i})\}.$$

The *scheduling problem* is to determine a valid schedule that minimize the total execution time $\mathtt{T}(\mathtt{s})$.

6.2 Runtime Estimation

The most important evaluation criteria of a parallel program is the total execution time, i.e., the time between start and termination of the program's computation. As defined above, this time consists of the time needed for task computation and the time needed for communication between tasks, i.e., for data (re)distribution. The task computation time depends on the algorithm implemented and on the data distribution used. Data distribution time depends on the number of processors and on the data distribution used. The choice of a suitable data distribution is an important issue of an efficient global execution time [60].

Data distribution time Technically, data distribution is accomplished by means of a (bijective) map from a global index set to a set of local index sets. In order to determine an optimal data distribution that leads to a minimal global execution time, we consider analitically derived execution time functions that include information about various data distributions as parameters.

For describing data distribution we adapt the *parameterized data distribution* proposed in [23] and [60]. According to this description, each data distribution of a $\mathtt{d}$ dimensional array of size $\mathtt{n_0} \times \ldots \times \mathtt{n_{d-1}}$ among a set of $\mathtt{p}$ processors can be described by a *distribution vector* of the form

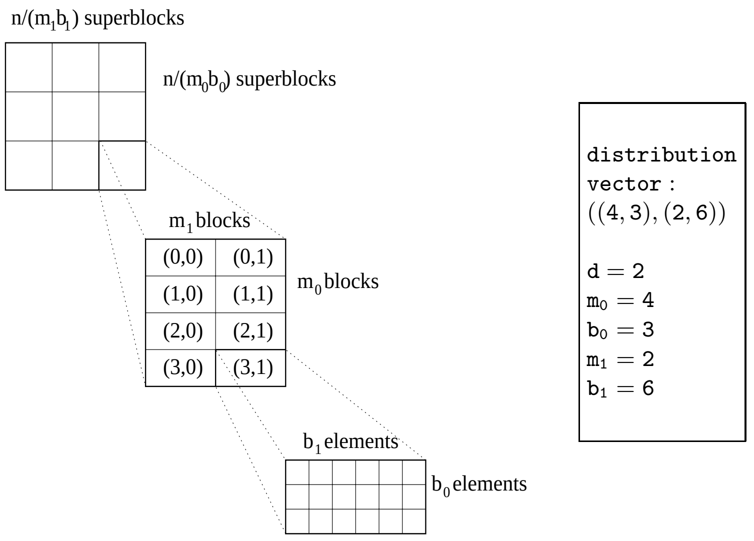


Figure 14: Distribution of an $\mathtt{n} \times \mathtt{n}$ array among 8 processors (example taken from [60]).

$$((\mathtt{m}_0, \mathtt{b}_0), \ldots, (\mathtt{m}_{d-1}, \mathtt{b}_{d-1}))$$

with $\mathtt{p} = \prod_{i=0}^{d-1} \mathtt{m}_i$ and $1 \leq \mathtt{b}_i \leq \mathtt{n}_i$. The value $\mathtt{m}_i$ specifies the number of processors in dimension $\mathtt{i}$. The value $\mathtt{b}_i$ specifies the block size in dimension $\mathtt{i}$. For an example see Figure 14.

Such a distribution vector describes, depending on the actual parameter values, various data distributions. For example, if $\mathtt{b}_i = 1$ for $\mathtt{i} = 1, \ldots, \mathtt{d} - 1$, then the vector describes a cyclic distribution, if $\mathtt{m}_i \cdot \mathtt{b}_i = \mathtt{n}_i$, the vector describes a block distribution.

Thus, we have a uniform description for various data distributions and we use this description to estimate the volume of data to be transferred for distribution. We assume that each one of the source processors communicates with each one of the destination processors of the distribution, i.e., we assume that data is evenly distributed among the processors and, thus, the communication costs are evenly distributed.

The communication itself is realized by single to single transfer, whose execution time depends on the parameters of the target machine model used: number of processors, startup time for point to point communication, byte transfer time, bandwidth of the interconnection network, see [12, 57, 34] for details.

In the examples presented, we used the runtime functions generated by [57] for prediction of communication time. These functions are efficient and provide accurate results for the examples used. As an example we present the runtime formula for MPI on the IBM SP2 for the single-transfer operation (the values $\mathtt{b}$ and $\mathtt{p}$ are the message size in bytes and the number of processors, respectively):

$$\mathtt{f}^{\mathtt{cf}}_{\mathtt{single}}(\mathtt{b}) \;=\; 211.8 \cdot 10^{-6} + 0.030 \cdot 10^{-6} \cdot \mathtt{b} \;\; [\mu\text{sec}].$$

The runtime functions generated in [34] are more precise but much more complex. They have to be used for finer granulated runtime predictions. As an example see below the runtime formula $\mathtt{f}^{\mathtt{GP}}_{\mathtt{single}}$ predicting the execution time of the single-transfer operation for MPI on the IBM SP2

$$\begin{aligned}\mathtt{f}^{\mathtt{GP}}_{\mathtt{single}}(\mathtt{p},\mathtt{b}) \; = \; & \mathtt{ln}(1 + ((11.339 \cdot 10^{-6} \cdot (\mathtt{b} + \sqrt{437.8 \cdot \mathtt{b}}) + \mathtt{e}^{\sqrt{\mathtt{b} \cdot 10^{-7}}} \cdot \\ & \mathtt{ln}(\mathtt{e}^{\sqrt[4]{b}} + 25.995) \cdot 0.010614) \cdot \mathtt{ln}(\mathtt{b} + 241.026) \cdot 10^{-6} \cdot \\ & (\mathtt{ln}(\mathtt{b}^{7/4} \cdot 148.41 \cdot 10^{-12} \cdot \mathtt{b} \cdot (\mathtt{b} + 10^{-6}) + 10) \cdot \\ & \mathtt{log}((\mathtt{b} \cdot 10^{-6} + 8103.083 + \mathtt{e}^{\sqrt[4]{b}}) \cdot (28 \cdot \mathtt{b}^2 + 1.0986) \cdot \mathtt{b} \cdot 4 + 4) \\ & \cdot 1096.6 \cdot (-13.81) \cdot 10^{-6} + 218.826)) \qquad [\mu\text{sec}]\end{aligned}$$

Computation time A composed module consists of tasks which represent basic modules, i.e., modules which are computed in a data parallel manner. Computation time of a basic module can be measured time, e.g., for already available library functions, or it can be an analytically derived estimation function. We use a runtime estimation model for SPMD programs in message-passing style [54]. Investigations for several applications from numerical analysis show that the runtime prediction formulas describe the execution time accurately enough to compare different execution schemes of the same application [60, 34].

6.3 Communication Costs

In this section we want to point out the communication costs associated to a task. In terms of the module graph, communication costs are costs of data operations (cf. Section 5.2) associated to data edges. For a data edge $(\mathtt{t_i}, \mathtt{t_j})$, communication costs are costs for (re)distributing data from a group of processors $\mathtt{p_i}$ to a group of processors $\mathtt{p_j}$.

Communication costs consist of costs for sending data and costs for receiving data. For a task $\mathtt{t_i}$, costs for sending data are associated to data edges $(\mathtt{t_i}, \mathtt{t_k})$ starting at $\mathtt{t_i}$, costs for receiving data are associated to data edges $(\mathtt{t_j}, \mathtt{t_i})$ ending at $\mathtt{t_i}$.

Several communication operations performed by the same processors are assumed to be worked off sequentially. $\mathtt{comm\_costs}((\mathtt{t_i}, \mathtt{t_j}))$ denotes the communication costs associated to edge $(\mathtt{t_i}, \mathtt{t_j})$ and $\mathtt{comm\_costs}(\mathtt{t_i}, (\mathtt{t_i}, \mathtt{t_j}))$ denotes the part of $\mathtt{comm\_costs}((\mathtt{t_i}, \mathtt{t_j}))$ associated to task $\mathtt{t_i}$. Thus, the communications costs associated to task $\mathtt{t_i}$ are:

$$\mathtt{comm\_costs}(\mathtt{t_i}) = \sum_{\mathtt{j}} \mathtt{comm\_costs}(\mathtt{t_i}, (\mathtt{t_j}, \mathtt{t_i})) + \sum_{\mathtt{k}} \mathtt{comm\_costs}(\mathtt{t_i}, (\mathtt{t_i}, \mathtt{t_k}))$$

Evaluation strategy In general, communication costs associated to an edge $(\mathtt{t_i}, \mathtt{t_j})$ may be added completely/partially to the source task $\mathtt{t_i}$ or/and to the end task $\mathtt{t_j}$, depending on the execution time evaluation strategy used. Our strategy is to add the communication costs completely to the source task, due to the following considerations:

- Experiments made did not notify significant differences between the two approaches: adding costs to source task or adding costs to receiving task. Figure 15 illustrates the results of the two strategies (denoted in the figure with `_source` and `_destination`, respectively) applied to a scheduling run for the CG method. The experiments were made by using the runtime formula for MPI on the IBM SP2 (cf. Section 6.2) for 4 processors, a vector size of 300, and a maximum of 400 generations.

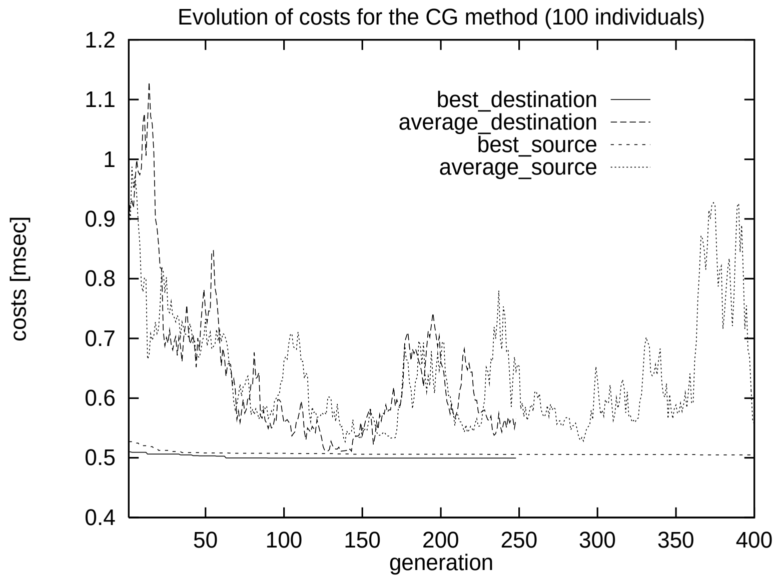


Figure 15: Evolution of costs.

For each one of the strategies applied, the figure shows the evolution of costs for the best individual found (denoted as `best_`) and the evolution of average costs of individuals of the population (denoted as `average_`). In this example, the difference between the final results of the two evaluation strategies, i.e., between the best individuals found, is very small, less that 2%. One of the runs stopped about 250 generations because no improvement was achieved in the last 100 generations.

- Independent of the strategy used, the receiving processors cannot start with the execution of a task until the data is received, i.e., until the communication time is expired.
- In the finally generated C+MPI program we use blocking send and receive operations, i.e., the send processors are blocked until the send buffer can be reclaimed. Similarly, the receive function blocks until the receive buffer actually contains the contents of the message.

6.4 Target Machine

Even if we did not specify it explicitly until now, all design decisions refer to a well defined target machine: a machine with a defined number of processors and with a specific runtime behaviour.

The information about the target machine is implicitly coded into the information used for the runtime evaluation:

- there is a given number of processors available,
- runtime formulas used for evaluating data transfer are derived from experiments made on the target machine,

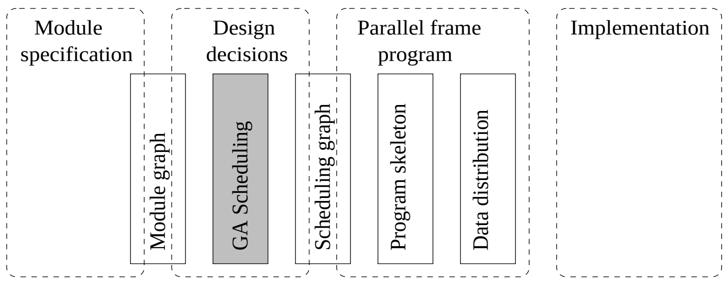
- analytically derived estimation function for the computation time are also derived from experiments made on a concrete machine, and
- run times for library functions are measured on a defined machine, too.

Thus, a solution for the scheduling problem is always a solution for the scheduling problem on a well defined target machine.

6.5 Section's Summary and Outlook

In this section we defined our scheduling problem (Section 6.1), which is enlarged in comparison with usual scheduling problems. Our scheduling problem is a multi-dimensional problem: we want to determine the execution order of tasks and assign groups of processors to tasks, but we also want to select implementation versions with appropriate data distribution, such that a minimal global execution time on a well defined target machine results. For this quite large optimization problem we use a genetic algorithm approach, an approach which we detail in the next section.

Additionally, we described in this section the functions used for estimating computation (Section 6.2) and communication costs (Section 6.3).



7 Genetic Algorithms

The search space for our scheduling problem is vast (as mentioned in Section 1, the problem is NP-complete [26]), a solution of the scheduling problem includes besides the execution order of the tasks, the processor groups to be assigned to each tasks, and for each task an individual version selected from a set of versions which differ by implementation, data distribution, number of processors used, and computation time consumed.

In this section we will describe how the scheduling problem can be solved by using a genetic algorithm (GA) approach: we give an overview of the GA paradigm, describe the problem's coding, and the functions used.

7.1 Overview on the Genetic Algorithm Approach

```
initialize(population)                      (cf. Section 7.3)
while not good_enough(fitness(population))  (cf. Section 7.6)
     select(population)
     mutation(population)                   (cf. Section 7.4)
     crossover(population)                  (cf. Section 7.5)
```

Figure 16: Overview on the genetic algorithm.

The usual form of genetic algorithms was described by Goldberg [33]. GAs are stochastic search techniques based on the mechanism of natural selection and natural genetics. They start (cf. Figure 16) with an *initial population*, i.e., an initial set of random solutions which are represented by chromosomes. The chromosomes evolve through successive iterations, called *generations*. During each generation, the solutions represented by the chromosomes of the population are evaluated using some measures of *fitness*. To generate the next generation, new chromosomes are formed. The chromosomes involved in the generation of a new population are selected according to their fitness values. Fitter chromosomes have higher probabilities of being selected. After several generations, the algorithm converges to the best chromosome which represents the *(sub)optimal solution* of the problem.

7.2 The Chromosomes

Chromosomes An individual (a *chromosome*) represents a solution of the scheduling problem by giving the relative execution order of the tasks, by selecting for each task an appropriate version, and by associating to each task a group of processors. A chromosome consists of 3 substrings of the same length: task part, version part, and processor part (details below). Three genes (`t`, `v`, `p`) belong together. The meaning is: a processor group `p` is available for the execution of version `v` of task `t`.

Example In Figure 17, the elements on position 3 belongs together, the meaning is: a processor group coded by 15, i.e., the processors 0, 1, 2, and 3, details below, is available for the execution of version 2 of task 5.

0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	(position)
0	2	6	5	4	3	7	1	(task part)
14	8	3	2	1	0	0	14	(version part)
15	15	1	15	2	8	4	15	(procs part)

Figure 17: A chromosome.

Genes Chromosomes are build up by *genes*. A gene of the *task part* corresponds to a node of the module graph. Due to technical reasons (cf. Section 5) this is not an one to one correspondence to modules of the module specification, but there is a bijective mapping between genes of the task part and nodes of the module graph.

Remember that there are dependencies between tasks expressing whether two tasks have to be executed sequentially (or can be executed concurrently), which have to be meet by a valid schedule. Thus, the tasks in a valid schedule are *topologically sorted* according to the module graph edges.

A gene of the *version part* represents a version of the corresponding task. Each task has one or more versions, e.g., different library functions, which may differ by the algorithm implemented, by the data distribution of parameters, or by the number of processors needed for execution.

A gene of the *processor part* codes the processors available for executing the corresponding task. We interpret the gene as a binary: a value 1 on position `i` means that processor `i` is available for the execution of the corresponding task.

Examples All examples in the next subsections are parts of chromosomes which code a schedule of the composed module from Figure 4. The task ids refer to the nodes of the module graph illustrated in Figures 12 and 13.

7.3 Chromosome Initialization Operation

The initialization operation is called at the beginning of the GA process and it generates the new individuals of the first population. The operation takes into consideration that a chromosome is build up of three distinguished parts. First, the task part is initialized, subsequently, the version part, and, last, the processor part genes get their values.

Initialization The genes of the task part are assigned randomly, each task appears once. Subsequently, in order to obtain a valid schedule, the tasks are topologically sorted according to the module graph edges. As an example see Figure 18; the corresponding module graph is illustrated in Figures 12 and 13.

```
randomly       3 | 19 | 0 | 15 | 16 | 17 | 6 | 8 | 13 | 7 | 4 | 18 |
                20 | 5 | 12 | 1 | 14 | 22 | 21 | 11 | 9 | 2 | 10

after sorting  0 | 21 | 2 | 5 | 4 | 6 | 3 | 14 | 12 | 13 | 7 | 8 |
                9 | 15 | 17 | 16 | 18 | 19 | 20 | 10 | 11 | 22 | 1
```

Figure 18: Task part genes.

A task description stores for each task an individual set of possible versions. For tasks corresponding to the same module the same versions (cf. Section 6.1) are available. In a second step, each gene of the version part gets a randomly selected value taken from the set of possible versions of the corresponding task.

The values of the processor part genes are assigned last. They have to fit with the corresponding task and version values. Some implementations, i.e., some task versions, predefine the number of processors needed, other implementations allow a varying number of processors. In this case, the number of processors is determined randomly. The group of processors used is initialized randomly, i.e., we determine randomly the processors and code the values binary.

Context dependence During the initialization operation there is a special handling for the tasks 0 and 1. These tasks are dependent from each other, both correspond to the composed module and represent the input and output node of the module graph, nodes $\mathtt{C}_0$ and $\mathtt{C}_1$, respectively. Between the two nodes there is a reciprocal dependence relation, i.e., they represent the same task and thus, selected version and number of processors used have to be the same for both. The relation is a context dependence relation, it was already introduced in Section 5.3.3. This relation implies some extra operation for the tasks during the GA process, too.

Experiments At the beginning of our experiments we generate each chromosome of the first population randomly. Experiments showed that the results are better if the initial population includes ”good” individuals, e.g., if a part of the initial population is taken from a previous (GA or another approach) scheduling.

7.4 Mutation Operation

The mutation operation generates a new individual by randomly changing a randomly selected individual of the population. The position where the operation is performed is randomly selected, too. There is no guarantee that a mutation operation really achieves a change. For some values there is no valid alternative. A scheduling chromosome fulfills some rules, e.g., tasks are topologically sorted, and it is not always possible to make a change at a randomly selected position without violating these rules. Different mutation operations are developed for different parts of the chromosome.

Processor part mutation randomly changes the group of processors assigned to a task. If the corresponding task version does not expect a predefined number of processors then, first, the number of processors is determined randomly. Figure 19 shows the mutation operation on processor part position 2. Version 3 of task 6 does not expect a predefined number of processors, so the number of processors, 2, is determined randomly. Then, we randomly select processors 1 and 2 (binary coded by 6).

position	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	
initial	0	2	6	5	4	3	7	1	(task part)
chromosome	14	8	3	2	1	0	0	14	(version part)
	15	15	1	4	2	8	4	15	(procs part)
resulted	0	2	6	5	4	3	7	1	(task part)
chromosome	14	8	3	2	1	0	0	14	(version part)
	15	15	6	4	2	8	4	15	(procs part)

Figure 19: Processor part mutation operation at position 2.

position	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	
initial	0	2	6	5	4	3	7	1	(task part)
chromosome	14	8	3	2	1	0	0	14	(version part)
	15	15	1	4	2	8	4	15	(procs part)
resulted	0	2	6	5	4	3	7	1	(task part)
chromosome	14	8	4	2	1	0	0	14	(version part)
	15	15	5	4	2	8	4	15	(procs part)

Figure 20: Version part mutation operation at position 2.

Version part mutation randomly chooses a new version, if available, of the corresponding task. Accordingly, the corresponding number of processors is adapted, if necessary. Figure 20 shows the mutation operation at version part position 2. The new version 4 of task 6 is randomly selected; this version needs 2 processors, so we randomly select processor 0 and 2 (binary coded by 5).

Task part mutation is more complex. The idea is to move a randomly selected task without violating task dependencies. According to that, we look for the set of neighbor tasks (neighborhood is to be seen inside the chromosome and not inside the module graph) which can be performed in parallel with the selected task and we try to change the relative order of these tasks by moving the selected task.

Let $\mathtt{t_i}$ be a task,

$$\mathtt{inlist(t_i)}=\{\mathtt{t_j};\ \mathtt{t_j} \text{ task with } (\mathtt{t_j},\mathtt{t_i}) \text{ module graph edge}\}$$

be the set of direct predecessors of task $\mathtt{t_i}$ in the module graph, and

$$\mathtt{outlist(t_i)}=\{\mathtt{t_j};\ \mathtt{t_j} \text{ task with } (\mathtt{t_i},\mathtt{t_j}) \text{ module graph edge}\}$$

be the set of direct successors of $\mathtt{t_i}$.

The *valid range* of a task $\mathtt{t_i}$ on position $\mathtt{i}$ in a chromosome is the set of positions $[\mathtt{pos_1}..\mathtt{pos_2}]$, $\mathtt{pos_1} \leq \mathtt{i} \leq \mathtt{pos_2}$, at which this task can be placed without violating dependence rules (because there are no precedence relations between $\mathtt{t_i}$ and the tasks on these positions). We define:

$$\mathtt{pos_1} = \begin{cases} 0, \text{ if } \mathtt{inlist(t_i)} = \phi \\ \mathtt{max}\{\mathtt{j};\ \mathtt{t_j} \in \mathtt{inlist(t_i)}\} + 1 \end{cases}$$

$$\mathtt{pos_2} = \begin{cases} \mathtt{maxposition}, \text{ if } \mathtt{outlist(t_i)} = \phi \\ \mathtt{min}\{\mathtt{j};\ \mathtt{t_j} \in \mathtt{outlist(t_i)}\} - 1 \end{cases}$$

In particular, the valid range starts behind all tasks which have an outgoing edge ending at the selected task and the valid range ends before any task having an ingoing edge which starts at the selected task. The valid range contains only tasks which have no dependence relation with the selected task and, thus, can be executed concurrently with it.

The task part mutation operation randomly moves the selected task within the valid range, corresponding versions and processor numbers are moved accordingly. In principle, each other permutation of the valid range elements may be used for this operation. The valid range of a task can consist of only one element (the initial position of the task) and the chromosome does not change. Figure 21 illustrates the task part mutation operation. The initial chromosome is on top, the result of the operation is on bottom. The valid range of task 3 from position 3 is between the positions 3 and 6, i.e., tasks 4, 5, and 6 can be executed concurrently with task 3. For a graphical illustration of the situation see Figure 12 and Figure 13. The new position 6 is randomly selected and the genes are moved accordingly.

Context dependence The mutation operation includes a special handling for the context dependent tasks. If a version or processor part mutation operation is performed on one of these tasks, then the counterpart is changed accordingly.

position	0	1	2	3	4	5	6	7	8	
initial	0	21	2	3	6	5	4	7	1	(task part)
chromosome	12	5	10	1	1	0	3	0	12	(version part)
	15	15	15	2	4	7	1	1	15	(procs part)
resulted	0	21	2	6	5	4	3	7	1	(task part)
chromosome	12	5	10	1	0	3	1	0	12	(version part)
	15	15	15	4	7	1	2	1	15	(procs part)

Figure 21: Task part mutation operation at position 3 (valid range is 3..6; randomly selected new position is 6).

7.5 Crossover Operation

The crossover operation generates a new individual by combining parts taken from two parent individuals. At first, the child is a copy of its first parent. Then we randomly select the crossover position and perform the actual crossover operation by introducing information from the second parent into the child chromosome. Different crossover operations are developed for different parts of the chromosome.

Processor and version part crossover substitute processor and version information of the child by the corresponding information of the second parent. In order to obtain a consistent individual, both corresponding information, i.e., version and processor number, are taken. This is not always mandatory, but we spare tests which otherwise would be necessary. The operations are performed starting at the selected position and continuing until the end of the chromosome is reached. Context dependent tasks, e.g., task 0 and 1, are kept consistent.

position	0	1	2	3	4	5	6	7	
first	0	2	3	5	4	6	7	1	(task part)
parent	22	12	0	2	0	0	0	22	(version part)
	6	7	7	6	7	7	1	6	(procs part)
second	0	2	6	5	3	4	7	1	(task part)
parent	14	8	3	2	0	1	0	14	(version part)
	15	4	1	7	9	5	2	15	(procs part)
child	0	2	3	5	4	6	7	1	(task part)
	14	12	0	2	1	3	0	14	(version part)
	15	7	7	7	5	1	2	15	(procs part)

Figure 22: Version part crossover operation at position 3.

Figure 22 shows a version part crossover operation at position 3. Starting at this position, the version and processor information from the second parent are copied into the child chromosome, i.e., the information for the tasks 5, 4, 6, 7, and 1. Due to the context dependence relation between task 0 and 1, the information of task 0 is also changed, accordingly to the new values corresponding to task 1.

Task part crossover puts the tasks of the child in the relative order of the tasks of the second parent. Only tasks between selected crossover position and end of the list are considered. For a consistent individual, the corresponding information, i.e., version and processor number information, are moved accordingly without changing the values. The positions of tasks 0 and 1 are predefined, i.e., 0 is the source of the module graph and consequently it has always position 0 in the chromosome, tasks 1 is the sink of the module graph and has always the last position in the chromosome. Thus, tasks 0 and 1 are not changed by this operation.

In Figure 23 the tasks 4, 6, 7, and 1 are put in the relative order of the second parent, i.e., 6, 4, 7, 1.

position	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	
first	0	2	3	5	4	6	7	1	(task part)
parent	22	12	4	2	2	0	0	22	(version part)
	10	15	2	9	8	4	1	10	(procs part)
second	0	2	6	5	4	3	7	1	(task part)
parent	21	18	4	0	2	4	0	21	(version part)
	15	13	2	7	5	2	3	15	(procs part)
child	0	2	3	5	6	4	7	1	(task part)
	22	12	4	2	0	2	0	22	(version part)
	10	15	2	9	4	8	1	10	(procs part)

Figure 23: Task part crossover operation at position 4.

7.6 Fitness Function

Fitness is the driving force of the Darwinian natural selection and, likewise, of GAs. It may be measured in many different ways. The accuracy of the fitness function with respect to the application is crucial for the quality of the results produced by the GA. Our fitness function evaluates the runtime of a program whose task schedule is represented by a chromosome. The costs include all computation and communication costs.

A chromosome codes unambiguously the relative order of tasks, their versions, and knows the processors used for each task. Task id, version number, and processors exactly define an implementation of a task with a well defined data distribution. Thus, computation and communication costs can properly be evaluated. Below we describe the general criteria for fitness evaluation.

7.6.1 General Criteria for Fitness Evaluation

Due to the structure of a schedule, the fitness evaluation process is a priori not uniquely determined. Therefore we made the following general arrangements. These arrangements have to be respected when evaluating the fitness of a chromosome. ($\mathtt{p_i}$ denotes the processor group associated to task $\mathtt{t_i}$.)

- The evaluation function considers tasks in the order they appear on the chromosome. Let $\mathtt{t_i}$, $\mathtt{t_{i+1}}$ be two tasks, then first the costs for task $\mathtt{t_i}$ are evaluated, before costs evaluation for task $\mathtt{t_{i+1}}$ is performed. The idea behind this procedure is that all tasks $\mathtt{t_j}$ which have to precede, according to the module specification, the current task $\mathtt{t_i}$ are already evaluated before starting the evaluation of $\mathtt{t_i}$.

- For each task $\mathtt{t_i}$, its input data has to be locally available, i.e., data transfers have to be completed, before the task execution starts:
  $\mathtt{start\_time(t_i)} \ \geq \ \mathtt{max}\{\mathtt{end\_time(t_j)}; \mathtt{i} > \mathtt{j}, (\mathtt{t_j}, \mathtt{t_i}) \ \mathtt{data\ edge}\}$

- Dependent tasks, i.e., tasks connected by module graph structure edges, are executed sequentially, i.e., a task has to complete its execution before the successor can be started:
  $\mathtt{start\_time(t_i)} \ \geq \ \mathtt{max}\{\mathtt{end\_time(t_j)}; \mathtt{i} > \mathtt{j}, (\mathtt{t_j}, \mathtt{t_i}) \ \mathtt{structure\ edge}\}$

- When starting the execution of a task $\mathtt{t_i}$, all processors $\mathrm{q_j}$ executing $\mathtt{t_i}$ have to be available and they start the execution at the same time:
  $\mathtt{start\_time(t_i)} \ \geq \ \mathtt{max}\{\mathtt{time(proc\ q_j\ available)};\ \mathtt{q_j} \in \mathtt{p_i}\}$

- All processors used for the execution of a task $\mathtt{t_i}$ finish the execution at the same time and are immediately available for other tasks.
  $\mathtt{time(proc\ q_j\ available)} = \mathtt{end\_time(t_i)} \ \mathtt{when} \ \mathtt{q_j} \in \mathtt{p_i}$

- Independent tasks executed by the same processors are executed sequentially, i.e., a task has to complete its execution before a successor can be started:
  $\mathtt{start\_time(t_i)} \ \geq \ \mathtt{max}\{\mathtt{end\_time(t_j)}; \mathtt{i} > \mathtt{j},\ \mathtt{p_j} \cap \mathtt{p_i} \neq \phi\}$

- There is no rule to be respected for independent tasks executed by different processors.

- Computation costs which accrue when executing a task $\mathtt{t_i}$ have to be considered when evaluating $\mathtt{t_i}$.

- Communications costs associated to a task $\mathtt{t_i}$ have to be considered when evaluating $\mathtt{t_i}$.

- Communication costs, i.e., costs for (re)distributing data from a group of processors $\mathtt{p_i}$ to a group of processors $\mathtt{p_j}$ are associated to data edges $(\mathtt{t_i}, \mathtt{t_j})$. In our implementation, these costs are completely added to the source task $\mathtt{t_i}$ (cf. Section 6.3). Generally, communication costs associated to a task $\mathtt{t_i}$ are:

$$\texttt{comm\_costs}(t_i) = \textstyle\sum_j \texttt{comm\_costs}(t_i,(t_j,t_i)) + \sum_k \texttt{comm\_costs}(t_i,(t_i,t_k))$$

where $\texttt{comm\_costs}((t_i,t_j))$ denotes the communication costs associated to data edge (t_i,t_j) and $\texttt{comm\_costs}(t_i,(t_i,t_j))$ denotes the part of $\texttt{comm\_costs}((t_i,t_j))$ associated to task t_i.

- Several communication operations performed by the same processor are assumed to be worked off sequentially.

7.7 Parameters of the Genetic Algorithm

It is possible to vary several parameters of a GA run:

- The *number of individuals* to be taken from a previous (GA or other) scheduling run. Experiments showed that we obtain better results (in fewer time and better optimizations), if we initialize the first population with some 'good' individuals at the beginning of the GA run.
- The *probability of mutation*, i.e., the probability of changing a selected individual by using the mutation operation.
- The *probability of crossover*, i.e., the probability of changing a selected individual by using the crossover operation.

Figure 24 shows the evolution of costs for various probability parameters, without initial individuals (`pc` denotes the probability of crossover, `pm` denotes the probability of mutation). Due to the problem's structure and the definition of the operations involved, a crossover operation causes greater changes in the average costs than a mutation operation. All runs found good scheduling solutions. We decided to use a fifty-fifty probability for mutation and crossover operations.

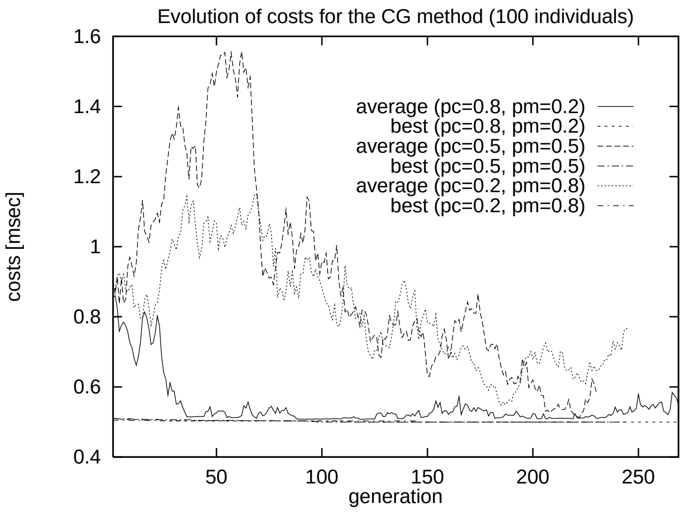


Figure 24: Evolution of costs by using various GA probability parameters.

7.8 Termination Step

The GA stops after having considered a predefined number of generation or if no improvement is obtained during a predefined number of generations. The runtime of the GA depends on the search space's size and on the problem size, see next section for more details.

7.9 Example of a GA Run

0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	(position)
0	21	2	4	6	3	5	7	10	12	11	15	(task part)
9	0	3	3	3	3	3	0	2	0	3	3	(version part)
15	15	15	1	2	4	8	15	15	14	1	2	(procs part)
12	*13*	*14*	*15*	*16*	*17*	*18*	*19*	*20*	*21*	*22*		(position)
13	8	14	16	17	19	9	18	20	22	1		(task part)
4	3	0	4	0	3	3	3	0	1	9		(version part)
12	2	12	12	13	2	8	4	15	15	15		(procs part)

Figure 25: A chromosome for the conjugate gradient iteration.

In this section we continue the CG example used until now. Figure 25 illustrates a chromosome for the CG iteration (cf. Figure 12 and 13 for the corresponding module graph). For each basic module there are various variants which basically differ in the number of processors, and on the data distribution used for each parameter. We used for our experiments 28 different versions for the composed module, 28 versions for the `mv_prod` module, 5 versions for each one of the other vector operations (`vv_prod`, etc., a total of 14 modules), and only one possibility for the scalar operations (`assign` and `op_assign`, 4 modules). The tasks representing the input and output nodes of the sequential loop depend from other tasks and therefore there are no variants to be specified for. Theoretically, there are $28^2 \cdot 5^{14}$ possibilities to combine the module versions. Additional possibilities are caused by different task permutations. The number of these permutations is determined by the number of operands of the $\|$ expressions; in our example there are $4! \cdot (2!)^6$ possibilities. The various processor groups which can be assigned to each one of the tasks causes another enlargement of the search space. For a target machine with `p` processors there are $\binom{\mathtt{p}}{\mathtt{q}}$ possibilities of selecting `q` processors, i.e., these possibilities are available for each task using `q` processors.

Despite the volume of the search space, the GA run shows a good convergence to an optimal solution. Figure 24 illustrates the evolution of costs during GA runs with 100 chromosomes per generation, maximal 300 generations, a problem size (vector size) of 300, and an assumed target machine with 4 processors. In Figure 24, the GA runs stop after about 250 generation because no improvement is achieved in the last 100 generations.

A (sub)optimal solution of the GA run usually evolves over several generations, it is the result of GA operations on randomly selected positions; it is unusual that a good individual occurs already in the first generation. In Figure 15 the evolution of costs of the best individual is good visible.

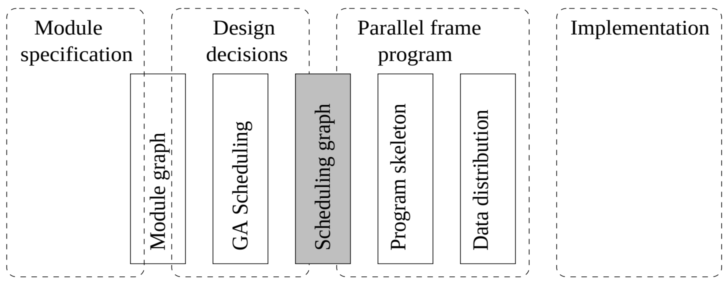
7.10 Section's Summary and Outlook

In this section we described how our scheduling problem can be solved by using a genetic algorithm approach. Therefore, we gave an overview of the genetic algorithm approach (Section 7.1), and we described the coding of our problem in terms of genetic algorithms, i.e., we coded tasks and their relative order, implementation versions, and processor groups assigned (Section 7.2). Subsequently, we described the process' initialization (Section 7.3), the mutation (Section 7.4), and the crossover (Section 7.5) operations used by the GA algorithm for constructing the next generation. The fitness operation (Section 7.6) evaluates the runtime of a program whose design decisions are represented by an individual of the GA process; the therefore general evaluation criteria used are described in Section 7.6.1. The GA process ends with the termination step described in Section 7.8, the process parameters are described in Section 7.7.

In this section we continued our conjugate gradient example, an individual describing a solution of the scheduling problem is given in Section 7.9.

In this section we solved the scheduling problem by using a genetic algorithm paradigm which offer the possibility of inspecting very large search spaces. Until now, scheduling algorithms do not deal with such multi-dimensional aspects of the problem. Usually, they are limited to search a relative execution order of tasks, and to assign to each task a group of processors, whereat the number of processors is predefined, except for malleable tasks [46, 62]. By means of the GA approach, we have the possibility to inspect large search spaces to find a solution of the problem. The result of a GA run represents a (sub)optimal solution of the scheduling problem, our fitness function evaluates the runtime, i.e., the total computation and communication time of the solution represented by the GA result. An individual, i.e., a result of the GA run, represents a solution of the scheduling problem by giving the relative execution order of the tasks, by selecting for each task an appropriate version with a well defined data distribution, and by assigning to each task a group of processors.

As next, we have to fix these implementation decisions into a parallel program (Section 9) which fits with the schedule found in the scheduling step. In order to do that, first we interpret the scheduling information represented by the scheduling result and store the scheduling information into a scheduling graph, a data structure where the scheduling information is easier accessible and which is used in subsequent steps to generate the parallel frame program. Beside this, the transformation into a scheduling graph eliminates redundancy, so that the quantity of information to be managed in the next steps is diminished, without loss of information.



8 Scheduling Graph

In this section we will interpret the result of the scheduling step and transform them into a scheduling graph, an intermediate data structure, where the scheduling information is directly accessible. The scheduling graph will be used for constructing the parallel frame program which fix all design decisions taken.

Scheduling graph The *scheduling graph* is an intermediate data structure which stores the relevant scheduling information and which is used for specifying the corresponding parallel frame program. The scheduling graph does not store the entire scheduling information, only elements relevant for constructing the skeleton of the parallel frame program are included. Some elements, e.g., data flow, are only partially considered, so that, for example, for adding the data distribution modules (cf. Section 11) we still go back to information stored in the module graph.

The information about the scheduling represented by a chromosome is stored into the chromosome, i.e., relative tasks order, implementation version corresponding to each task, and group of processors assigned to each task. Information stored in a chromosome is often redundant, see, e.g., a situation like the following one: let `A`, `B`, and `C` be tasks and the information is "`A` has to be performed before `B` and before `C`, and `B` has to be performed before `C`". The statement "`A` has to be performed before `B` and `B` has to be performed before `C`" furnish, from the scheduling point of view, the same information by using only 2 instead of 3 relations between tasks. Our algorithm for constructing the scheduling graph does not eliminate redundancy completely but it reduce it considerably.

The procedure to obtain a program representing the solution of the GA scheduling run consists of several steps. First of all, the scheduling graph is constructed, and the algorithm which transforms the scheduling result into a parallel program works on this data structure.

8.1 Constructing the Scheduling Graph

Below we use the notion of *task* in the meaning of "task corresponding to a gene of the task part of the chromosome" and we use for short *task* $\mathtt{t_i}$ for "task corresponding to the `i`-th gene of the task part of the chromosome" and *processor group* $\mathtt{p_i}$ for "processor group corresponding to the `i`-th gene of the processor part of the chromosome".

Nodes of the scheduling graph correspond to genes of the task part of the chromosome, the correspondence is accomplished by means of a bijective map. Thus, scheduling graph and module graph (Section 5) have the same node sets. *Annotations* attached to nodes record additional information concerning the scheduling, e.g., the group of processors assigned.

Edges The scheduling graph is a directed acyclic graph. The *edges* of the scheduling graph express the relative order in which the tasks have to be executed. The meaning of an edge $(\mathtt{t_i},\mathtt{t_j})$ is that task $\mathtt{t_i}$ has to be executed and its execution has to be finished before execution of task $\mathtt{t_j}$ is started. There are three steps for constructing the scheduling graph edges, the steps are performed sequentially, one after the other.

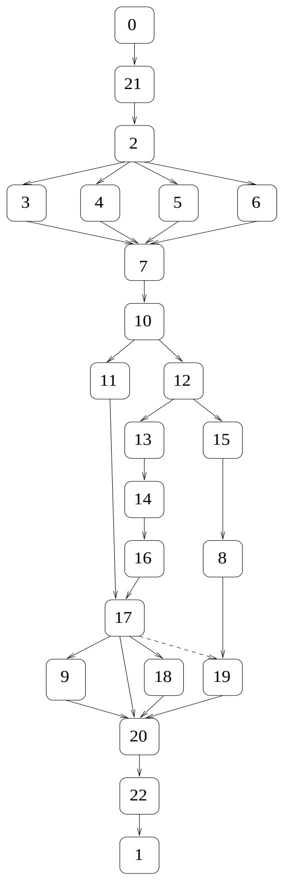
> **Step 1**: Edges expressing a sequential execution order caused by the use of common processors are constructed.
> Let $\mathtt{t_i}$ be a task which uses processor group $\mathtt{p_i}$. We compute for each processor $\mathtt{p} \in \mathtt{p_i}$ a task $\mathtt{t_{j_p}}$, with $\mathtt{j_p} = \mathtt{max}\{\mathtt{k}, \mathtt{k} = \mathtt{0..i} - \mathtt{1}, \mathtt{t_k}$ uses processor $\mathtt{p}\}$, being the last task which uses processor p before task $\mathtt{t_i}$ uses it. If there is no path $(\mathtt{t_{j_p}}, \ldots, \mathtt{t_i})$ into the scheduling graph, then add edge $(\mathtt{t_{j_p}}, \mathtt{t_i})$ to the scheduling graph.

Most of the sequential order dependencies are covered by edges constructed in this first step. Beside these dependencies, there are dependencies between tasks executed by disjoint groups of processors but nevertheless executed sequentially.

> **Step 2**: Edges expressing a sequential execution order caused by the structure of the initial module expression are constructed.
> Let $(\mathtt{t_i}, \mathtt{t_j})$ be a structure edge of the initial module graph (cf. Section 5). If there is no path $(\mathtt{t_i}, \ldots, \mathtt{t_j})$ into the scheduling graph, then add edge $(\mathtt{t_i}, \mathtt{t_j})$ to the scheduling graph.
>
> **Step 3**: Analogously to the second step, edges expressing a sequential execution order caused by the data flow are constructed.

In principle it is not necessary to separate step two and step three. We make this distinction, and favor herewith structure edges, in order to obtain a module expression with more similarity with the initial module specification. This effect is a help for the user when checking the results of the scheduling step.

Example As an example see the scheduling graph in Figure 26. On the right, we specify the processors assigned to each task. The task order used is the relative task order specified by the scheduling (cf. the chromosome in Figure 25). All graph edges, except edge (17,19), are added during the first construction step. Edge (17,19) is added in the second step because in the module graph there is a structure edge between the nodes 17 and 19 (cf. module graph in Figure 12) and there is no path (17,...,19) into the scheduling graph. In this example there is no edge added in the third construction step.



task		processors
0	←	{0,1,2,3}
21	←	{0,1,2,3}
2	←	{0,1,2,3}
4	←	{0}
6	←	{1}
3	←	{2}
5	←	{3}
7	←	{0,1,2,3}
10	←	{0,1,2,3}
12	←	{1,2,3}
11	←	{0}
15	←	{1}
13	←	{2,3}
8	←	{1}
14	←	{2,3}
16	←	{2,3}
17	←	{0,2,3}
19	←	{1}
9	←	{3}
18	←	{2}
20	←	{0,1,2,3}
22	←	{0,1,2,3}
1	←	{0,1,2,3}

Figure 26: A scheduling graph for the conjugate gradient iteration (cf. Figure 25 for the corresponding chromosome). The right figure shows the processor group assigned.

The following lemma proves that the scheduling graph information respects the scheduling expressed by a chromosome.

Lemma 1 (Scheduling graph information)
Let $\mathtt{t_i}$*,* $\mathtt{t_j}$*,* $\mathtt{i} < \mathtt{j}$*, be genes of the task part of a chromosome. If task* $\mathtt{t_i}$ *has to be executed before task* $\mathtt{t_j}$*, then there is a path* $(\mathtt{t_i}, \ldots, \mathtt{t_j})$ *inside the scheduling graph.*

Proof: From the scheduling point of view there are several reasons why a task has to be executed before another task: there is a data flow between the tasks, there are common processors assigned to the tasks, or the tasks use other common resources. We prove the issues one by one.

1. If there is a data flow between the tasks $\mathtt{t_i}$, $\mathtt{t_j}$, then there is a data edge $(\mathtt{t_i}, \mathtt{t_j})$ in the module graph. Then the lemma's statement follows according to the third construction step.

2. Let $\mathtt{p_i}$, $\mathtt{p_j}$ be the processor groups assigned to $\mathtt{t_i}$, $\mathtt{t_j}$, respectively. Let $\mathtt{p} \in \mathtt{p_i} \cap \mathtt{p_j}$ be a processor assigned to both tasks. We can demonstrate by

induction using the scheduling graph's first construction step that there is either an edge $(\mathtt{t_i}, \mathtt{t_j})$ or a path $(\mathtt{t_i}, \ldots, \mathtt{t_j})$ in the scheduling graph.

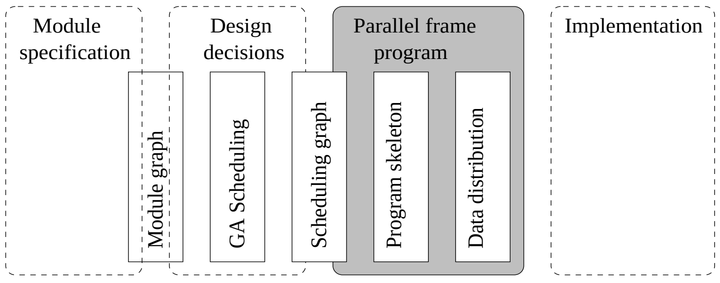
3. If the tasks $\mathtt{t_i}$, $\mathtt{t_j}$ use common resources, then this fact has to be specified initially in the module expression by means of a $\circ$ operator (see Section 4) and, subsequently, in the module graph by a structure edge (see Section 5). The statement of the lemma results from the second construction step.

□

8.2 Section's Summary and Outlook

We concentrated the design decisions taken by the scheduling step into a scheduling graph where the information is easier accessible and where redundancy is mostly eliminated. Subsequently, we proved the correctness of the scheduling graph, i.e., we proved that the sequential order of tasks is conserved by the scheduling graph.

The next sections will prove that the scheduling graph stores the information necessary for constructing a parallel frame program, i.e., the next sections show how the parallel frame program can be constructed by using the scheduling graph information. The parallel frame program generation takes place in three steps: first, we generate a program skeleton, subsequently program fragments dealing with data distribution are added, and, finally, the communication environment is established.



9 Parallel Frame Program

A *parallel frame program* express the degree of parallelism that should be exploited for a given distributed memory machine. A parallel frame program encloses beside a specification of the method to be implemented, all necessary design decisions. Information about the structure of the method to be implemented was specified in the module specification. We obtain the parallel frame program by fixing distribution of variables, scheduling of modules, and assignment of groups of processors to module activations. This information was computed in the design decision step.

We divide the generation of the parallel frame program into three subsequent steps: first, we generate a program skeleton, then we add data distribution operations, and, in a third step, we add operations to manage the communication domains. The program construction steps are described in Section 10 and 11. In this section we introduce the syntax of the language used.

9.1 Syntax of a Parallel Frame Program

A parallel frame program has a structure which is similar to a module specification (cf. Section 4) but contains additional information about data distribution, assignment of processors to module activations, and about communication environment. Figure 27 shows the language's syntax. We will explain some details on the example from Figure 28 which shows a fragment of a parallel frame program for the conjugate gradient iteration implemented with 4 processors. The complete parallel frame program can be found in Section 12.

In this example, for a better readability, we leave out all communication domain operations, i.e., generation/destruction of communicators and data transfer operations, and show only a rough structure of the program. Additionally, in order to distinguish several calls of the same module, we indexed them. This feature is not part of the parallel frame program. In Figure 28 the four `vv_prod` calls (lines 05 to 08) and some scalar operations (lines 09 to 12) are executed concurrently, the remainder of the operations are pure data parallel calls executed sequentially. The parameters' data distribution, e.g., denoted as $\mathtt{vec}(4,75)$, is indicated for each module call within the parameter list. For details on the parameterized data distribution see Section 6.2. Data distribution usually refers to the processor group assigned to the module activation. For particular parameters it is also possible to indicate a processor group which differs from the processor group assigned to the module, e.g., in line 01 scalar λ_0 is located

```
pfp_program →      program identifier pfp_module_set
pfp_module_set →   pfp_module pfp_module_set
                   | ε
pfp_module →       module module_decl { statements }
module_decl →      identifier ( parameters )
parameters →       param param_list   |  ε
param_list →       , param param_list   |  ε
param →            param_type identifier : data_descr
                   | comm_descr
                   | proc_group
param_type →       in | out | ε
data_descr →       data_type ( number_list ) data_distrib
data_type →        struct_type elem_type
struct_type →      vec | mat | ε
elem_type →        char | int | float | double
data_distrib →     :  proc_group   |  ε
c_parameters →     c_param c_param_list   |  ε
c_param_list →     , c_param c_param_list   |  ε
c_param →          identifier c_param_descr
                   | proc_group
c_param_descr →    : ( number_list ) c_data_distrib   |  ε
c_data_distrib →   :  proc_group   |  ε
proc_group →       proc ( number_list )
comm_descr →       comm identifier
number_list →      number | number_list , number
statements →       stmt stmt_list
stmt_list →        op_sign stmt stmt_list
                   | ε
op_sign →          ||  |  ∘
stmt →             identifier ( c_parameters )
                   | { statements }
                   | for ( enum_expr , comm_descr ) statements
                   | parfor ( enum_expr , comm_descr ) statements
                   | if ( simple_expr , comm_descr ) statements
simple_expr →      identifier relop identifier
                   | ε
enum_expr →        identifier = bound ..  bound
                   | simple_expr
bound →            identifier | number
comments →         [ text without brackets ]
```

Figure 27: Syntax of the parallel frame program.

only on processor 3. This kind of processor specification is mostly used for parameters of data distribution modules (for an example confer Section 11).

Usually, the processor group assigned to a module is not directly indicated. For each module we indicate by means of the `comm` parameter the communicator (cf. Section 11.2) used for the communication inside the module. A communicator specifies implicitly the set of processors concerned. An exception are

```
(01) module CG (in A:mat double(4,75,1,1), p:vec double(4,75),
         $p_0$:vec double(4,75), w:vec double(4,75), $w_0$:vec double(4,75),
         r:vec double(4,75), $r_0$:vec double(4,75), x:vec double(4,75),
         $\lambda_0$:double:proc(3), out p:vec double(4,75),
         x:vec double(4,75), r:vec double(4,75), comm $c_{CG}$[0,1,2,3])
(02) {
(03)  for (k=0..K,  $c_{for}$[0,1,2,3])
(04)    { mv_prod ([in] A:(4,75,1,1), p:(4,75), [out] w:(4,75), $c_{mv\_prod}$
                                                            [0,1,2,3])
(05)      ∘ { vv_prod$_1$ ([in] p:(1,300), $r_0$:(1,300), [out] tmp$_1$, proc(0))
(06)          ‖ vv_prod$_2$ ([in] w:(1,300), p:(1,300), [out] $\lambda_1$, proc(2))
(07)          ‖ vv_prod$_3$ ([in] w:(1,300), w:(1,300), [out] tmp$_2$, proc(1))
(08)          ‖ vv_prod$_4$ ([in] w:(1,300), $w_0$:(1,300), [out] tmp$_3$, proc(3))
          }
(09)      ∘ {  op_assign$_1$ ([in] tmp$_1$, op, $\lambda_1$, [out] $\xi$, proc(2))
(10)          ‖ { op_assign$_2$ ([in]tmp$_3$, op, $\lambda_0$, [out] $\nu$, proc(3))
(11)              ∘ assign ([in] $\lambda_1$, [out] $\lambda_0$, proc(3)) }
(12)          ‖ op_assign$_3$ ([in]tmp$_2$, op, $\lambda_1$, [out] $\mu$, proc(1))
          }
(13)      ∘ sv_prod$_1$ ([in] $\xi$, p:(4,75), [out] tmp$_4$:(4,75), $c_{sv\_prod_1}$[0,1,2,3])
(14)      ∘ vv_assign ([in] w:(1,300), [out] $w_0$:(1,300), proc(3))
(15)      ∘ sv_prod$_2$ ([in] $\xi$, w:(4,75), [out] tmp$_5$:(4,75), $c_{sv\_prod_2}$[0,1,2,3])
(16)      ∘ sv_prod$_3$ ([in] $\mu$, p:(4,75), [out] tmp$_6$:(4,75), $c_{sv\_prod_3}$[0,1,2,3])
(17)      ∘ sv_prod$_4$ ([in] $\nu$, $p_0$:(4,75), [out] tmp$_7$:(4,75), $c_{sv\_prod_4}$[0,1,2,3])
(18)      ∘ vv_sub ([in] r:(4,75), tmp$_5$:(4,75), [out] r:(4,75), $c_{vv\_sub}$)
...       ...
```

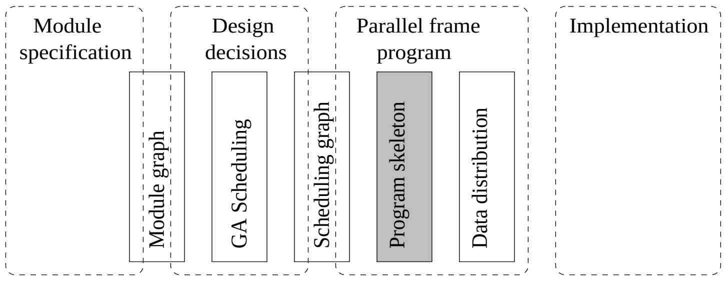
Figure 28: Fragment of a parallel frame program for the conjugate gradient iteration with 4 processors. For the present, all communication domain management operations are left out. Brackets [...] include comments, indexes in the module names are for a better overview only.

modules processed by a single processors, e.g., the vv_prod modules in line 05 to 08. For these modules we indicate by means of the proc parameter the processor assigned.

9.2 Section's Summary and Outlook

In this section we specified the syntax of the language used for parallel frame programs. The language has a similar structure as the module specification language, but it is augmented by elements for specifying implementation relevant information, i.e., with this language elements it is possible to fix the design decisions taken.

The next sections will describe how to obtain the parallel frame program from the scheduling graph constructed in Section 8. The construction of a parallel frame program includes construction of a program skeleton, insertion of data distribution modules, and communication management.



10 Transforming Scheduling Graphs into Parallel Frame Programs

The scheduling graph records the relative order which have to be meet by a parallel frame program without specifying an exact execution order. Our goal is to specify a parallel frame program which meets the relative order of the tasks and which has a structure as compact as possible.

The first step when transforming scheduling graphs into parallel frame programs is the construction of a program skeleton. This section contains the technical details concerning division of program construction in manageable parts and connection of generated program fragments. Beside this, we also demonstrate the correctness of the program skeleton constructed, i.e., we demonstrate that the program meets the design decisions taken before.

The general structure of the procedure for constructing parallel frame programs is shown in Figure 29. The procedure is recursive, we construct parallel frame programs for subgraphs of the scheduling graph and compound resulting program fragments according to composition rules. For a better readability, we omit brackets in the constructed program as soon as possible, the operators used are assumed to be left associative.

```
construct_program (graph G)
{ while (node $t_i \in$ G not yet processed)        (cf. Section 10.2)
    select_subgraph G($t_i$);                        (cf. Section 10.1)
    P(G($t_i$)) = construct_program(G($t_i$));       (cf. Section 10.3)
    connect P(G($t_i$)) to already available         (cf. Section 10.3)
                          program fragments;
}
```

Figure 29: Overview on constructing parallel frame programs.

10.1 Selecting Scheduling Subgraphs

The first step when constructing a parallel frame graph from a scheduling graph is to select the working area (cf. Figure 29), i.e., to specify the subgraph to be transformed. A subgraph of the scheduling graph is specified by a subset of nodes. Only nodes of this subset are processed by the current procedure. The procedure ends when all nodes of the current subgraph are processed.

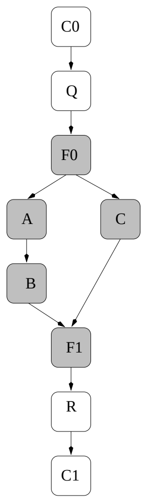


Figure 30: Scheduling graph with selected subgraph.

The specification of subgraphs depends on the type of nodes involved. A subgraph for a composed module, as well a subgraph for other block structures with a well defined begin and end node, e.g., loops or conditionals, contains the nodes corresponding to tasks inside the structure as well as the structure's input and output node. Generally, a scheduling graph corresponding to a *block structure* contains the nodes which can be reached when starting at the input node and from where the output node of the block can be reached. In order to construct the corresponding parallel frame program, the transformation process starts at the structure's input node. In Figure 30 the shaded nodes specify a scheduling subgraph for a sequential loop with input node $\mathtt{F}_0$ and output node $\mathtt{F}_1$ (cf. Figure 9); the transformation process starts at $\mathtt{F}_0$.

The scheduling subgraph corresponding to a composed module is identical with the initial scheduling graph. The starting node for the transformation process is the node corresponding to the input node of the composed module, e.g., in Figure 30 node $\mathtt{C}_0$.

10.2 Passing through Scheduling Subgraphs

When specifying the subgraph, the starting node for the process is specified, too. Beginning at this node, the transformation function passes the scheduling subgraph and constructs the corresponding parallel program fragments. Subsequently, these program fragments are composed to a parallel frame program. The general construction rules are described below, in Section 10.3 are some additional details.

We do not have a priori any information about the structure of the scheduling graph, except the fact that it is an acyclic graph. In the examples in this and in the next section we will illustrate that the scheduling graph has a complex structure and it cannot be traversed straight ahead. The process often stops and resumes.

Rules for passing through the graph Below, we describe the rules for passing the graph and proceeding nodes. An explanatory example follows.

1. Going out from a node `t_i`, the process *visits all its neighbors* connected via outgoing edges.
2. The graph is *passed top down* and neighbor nodes are processed in lexicographical order.
3. A *node* `t_i` *is processed* only if it has *a single predecessor* and the node was called by this predecessor, or if the processes for all its direct predecessor nodes *are already completed*.

The consequence of these rules is that the pass through the graph is often interrupted, but there are well defined rules for the process' resumption.

Constructed program fragment

{ `P(3)` || `P(4)` } ∘ { `P(5)` || `P(6)` }

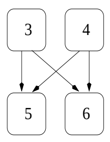
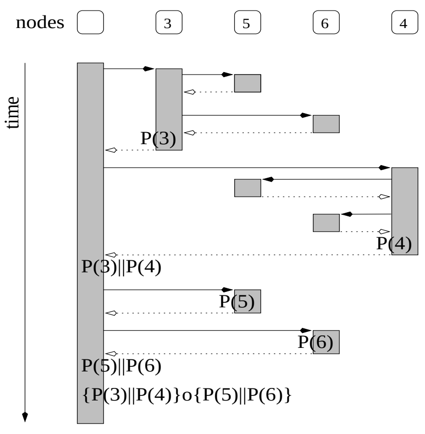


Figure 31: Scheduling graph (cross-joint structure).



Constructed program fragment

{ `P(3)` || `P(4)` }
∘{ `P(5)` || `P(6)` }

Figure 32: Procedure call sequence for the scheduling graph from Figure 31 (program fragments are indicated where they are generated).

Example We use Figure 31 and Figure 32 to explain the rules specified above. In Figure 31, when node 5 is processed during the first visit of this node, i.e., when coming from the procedure which process node 3, then we obtain the program fragment { `P(3) ∘ P(5)`} || `P(4)`, which is not correct because `P(4)` and `P(5)` have to be performed sequentially. If node 5 is processed during the second visit of this node, i.e., when the call for the procedure for node 5 comes from the procedure which process node 4, then we obtain the program fragment `P(3)` ∥ {`P(4)` ∘ `P(5)`}, which is also not correct because `P(3)` and `P(5)` have to be performed sequentially. Thus, after proceeding the nodes 3 and 4, the process has to be interrupted and it has to be restarted with the nodes 5 and 6. The corresponding program fragments, `P(5)` and `P(6)`, have to be performed sequentially to the program fragments `P(3)` and `P(4)`, but `P(5)` and `P(6)` are not dependent from each other and can be performed in parallel. Figure 31 shows the program constructed, Figure 32 shows the procedure calls used for constructing this program.

Rules for resuming the process

1. A *process can be restarted* with nodes ready to be processed. Ready nodes are nodes of the current subgraph whose predecessors, i.e., *all* their predecessors, are already processed.
   In our example, when the process is interrupted after nodes 3 and 4 are processed, then the nodes of 5 and 6 fulfill this property, they are ready.

2. The process is *restarted over and over* with ready elements, until all ready nodes are processed.

3. It is obvious that there are no edges between nodes of a ready set and, thus, there is no dependence between the program fragments corresponding to these nodes. These program fragments can be *performed concurrently.*
   Example: in Figure 31, `P(5)` and `P(6)` are performed in parallel.

4. When we restart an interrupted process, then the constructed program fragments are dependent from the program fragments constructed before, thus the program fragments have to be *performed sequentially.*
   Example: in Figure 31, `P(5)` ∥ `P(6)` is performed sequentially to the program fragment constructed before.

Constructed program fragment
(by using the open chain rule)

```
P(2)
∘ P(3)
∘ { P(4) ∥ P(5) ∥ P(6) }
∘ P(7)
```

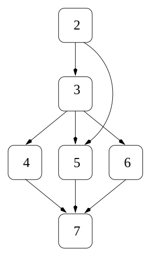


Figure 33: Scheduling graph (fork-joint structure with a bridge edge).

Next, we regard the example illustrated in Figure 33. When following the rules described above, we obtain the program

`P(2)` ∘ `P(3)` ∘ {`P(4)` ∥ `P(6)`} ∘ `P(5)` ∘ `P(7)`

although there is no dependence between `P(4)`, `P(5)`, and `P(6)`. The corresponding procedure calls are illustrated in Figure 34. The problem is that when

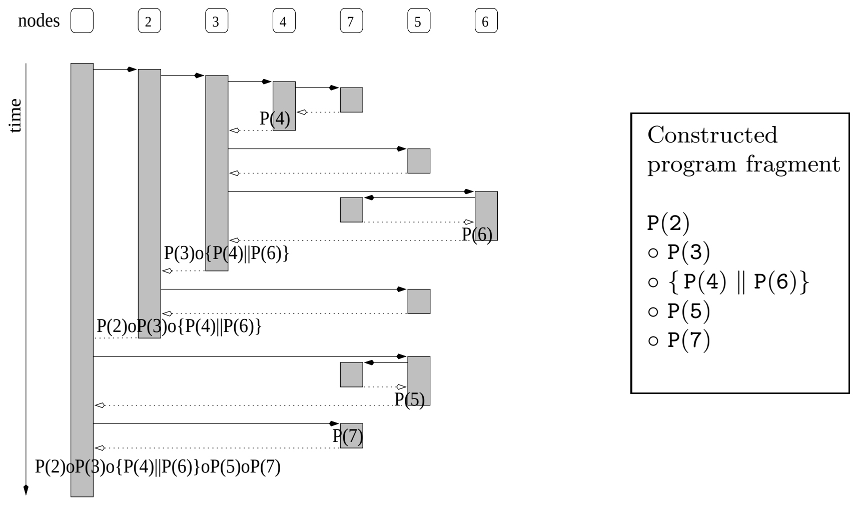


Figure 34: Procedure call sequence for the scheduling graph from Figure 33 without using the open chain rule (program fragments are indicated where they are generated).

proceeding the nodes 4, 5, and 6, node 5 has an additional direct predecessor (node 2) which is still in work, i.e., the procedure is not yet finished because the process is working top down and it completes a node only after all son nodes have been processed. In order to obtain an efficient program code which represents the actual dependencies, we adapt the rules specified above and allow nodes in particular cases to be processed, although not all their direct predecessors are completed.

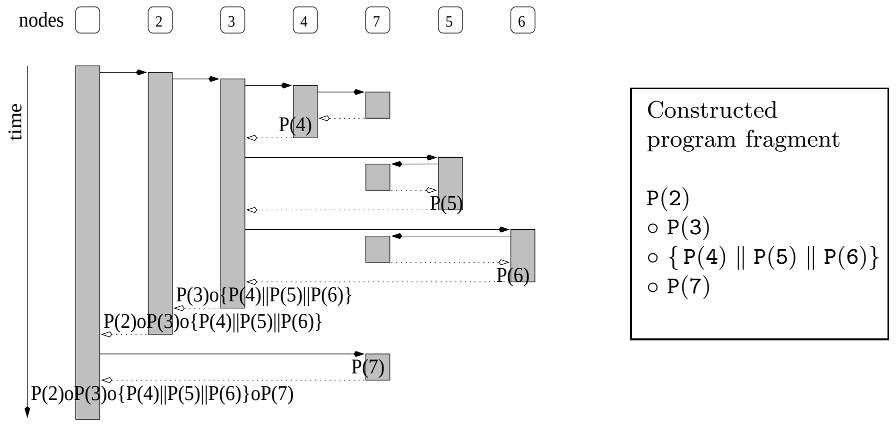


Figure 35: Procedure call sequence for the scheduling graph from Figure 33 when using the open chain rule (program fragments are indicated where they are generated).

Open chain rule A node with more than one predecessor is processed if all its predecessors are still in work when handling the affected node and if all its predecessors are not yet completed, i.e., the procedures for its predecessors are in a chain of "open" recursive procedure calls.

For the scheduling graph from Figure 33 we illustrate in Figure 35 this chain of open procedure calls: when node 5 is called, then the procedures for its predecessors, i.e., 2 and 3, are still open. Figure 35 illustrates procedure calls and program fragments constructed by using the open chain rule.

Lemma 2 (Correctness of open chain rule)
The open chain rule preserves the correctness of the resulting program, the parallel frame program constructed by means of this rule still fulfills the scheduling rules defined by the scheduling graph.

Proof: Due to the fact that all these nodes, i.e., the affected node and its "open" predecessors, are on a path connected by graph edges, the program fragment for the affected node is connected via $\circ$ operators (cf. Table 1 in Section 10.3) to the program fragments of its predecessors. Furthermore, the order of the program fragments is given by the order of the nodes in the scheduling graph. Thus, the resulting program meets the scheduling rules expressed by the scheduling graph.

$\square$

Note 1 A situation like in Figure 33 occurs due to the sequential construction algorithm used for the scheduling graph (cf. Section 8). We obtain such a graph, e.g., if `(2,3)`, `(2,5)`, `(3,4)`, and `(3,6)` are edges caused by the use of common processors and thus inserted first into the scheduling graph and `(3,5)` is a structure or a data edge of the module graph and an edge `(3,5)` is added to the scheduling graph during the second or third construction step because until this time there is no path `(3,...,5)` into the scheduling graph.

Note 2 The situation illustrated in Figure 33 cannot occur with a node which is an input/output node of a structure, i.e., node 3 cannot be an input node of a loop or another block structure. The reason is that all edges caused by the use of common processors and all structure edges from outside a loop pass across the input/output node and they cannot bridge such a node.

10.3 Constructing and Connecting Program Fragments

The procedure for constructing parallel frame programs passes through the scheduling graph and constructs program fragments for subgraphs. Table 1 shows the program construction rules for some subgraphs.

Nodes of a scheduling graph mostly represent activations of modules specified in the initial module specification. Additionally, there are 'technical' nodes conditioned by construction steps, e.g., input/output nodes for composed modules and for loops. The construction process for the parallel frame program has to consider this fact. In this section we show which program fragments are constructed for each particular node type.

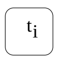
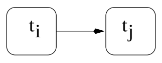
G	$P(G)$
	• t_i module node and $\#p_i > 1$ $\texttt{Create\_domain}\,(\textbf{proc}\ (p_i),\ c_{p_i})$ $\circ\ M_{t_i}(x_1 : d_{t_i}^{x_1}, \ldots, x_{n_{t_i}} : d_{t_i}^{x_{n_{t_i}}},\ c_{p_i})$ $\circ\ \texttt{Free\_domain}\,(c_{p_i})$ • t_i module node and $\#p_i = 1$ $M_{t_i}(x_1 : d_{t_i}^{x_1}, \ldots, x_{n_{t_i}} : d_{t_i}^{x_{n_{t_i}}},\ \textbf{proc}\ (p_i))$ • t_i loop input node and $\#p_i > 1$ $\texttt{Create\_domain}\,(\textbf{proc}\ (p_i),\ c_{p_i})$ $\circ\ L_{t_i}(i = 0, \ldots, n,\ c_{p_i})$ $\{$ • t_i loop input node and $\#p_i = 1$ $L_{t_i}(i = 0, \ldots, n,\ \textbf{proc}\ (p_i))$ $\{$ • t_i conditional input node and $\#p_i > 1$ $\texttt{Create\_domain}\,(\textbf{proc}\ (p_i),\ c_{p_i})$ $\circ\ I_{t_i}(\ldots,\ c_{p_i})$ $\{$ • t_i conditional input node and $\#p_i = 1$ $I_{t_i}(\ldots,\ \textbf{proc}\ (p_i))$ $\{$ • t_i loop or conditional output node and $\#p_i > 1$ $\}$ $\circ\ \texttt{Free\_domain}\,(c_{p_i})$ • t_i loop or conditional output node and $\#p_i = 1$ $\}$ • t_i composed module input node $M_{t_i}(x_1 : d_{t_i}^{x_1}, \ldots, x_{n_{t_i}} : d_{t_i}^{x_{n_{t_i}}},\ c_{p_i})$ $\{$ • t_i composed module output node $\}$
	• t_i, t_j module nodes $\{P(t_i)\} \circ \{P(t_j)\}$ • t_i input node or t_j output node $P(t_i)\ P(t_j)$ • t_i output node and t_j module node $P(t_i) \circ \{P(t_j)\}$ • t_i module and t_j input node $\{P(t_i)\} \circ P(t_j)$ • t_i output node and t_j input node $P(t_i) \circ P(t_j)$

Table 1: Transforming scheduling graphs into parallel frame programs.

In Table 1, let $\mathtt{t_i}$ be a node of a scheduling graph, $\mathtt{P(t_i)}$ denotes the parallel program fragment corresponding to $\mathtt{t_i}$. $\mathtt{M_{t_i}}$ is the module activation corresponding to $\mathtt{t_i}$, $\mathtt{d^x_{t_i}}$ denotes the data distribution of variable $\mathtt{x}$ in module activation $\mathtt{M_{t_i}}$, $\mathtt{p_i}$ specifies the group of processors mapped to $\mathtt{t_i}$, and, thus assigned to execute the corresponding program fragment, and $\mathtt{c_{p_i}}$ denotes a communication environment including all processors of group $\mathtt{p_i}$.

Module activations The parallel frame program for a node $\mathtt{t_i}$ representing a module activation realizes a module call. In comparison with the initial module specification (cf. Section 4), the module call contains additional data computed by the design decisions step, i.e., a reference to the group of processors to be used for executing the module call, and the parameters' actual data distribution. An example, but without communication domain management operations, can be seen in Figure 28, line 04, a call of `mv_prod`(...). The parameters' data distribution specified refers to the group of processors assigned for executing the module activation. The group of processors is indirectly specified by means of the communicator specification, e.g., $\mathtt{comm\ c_{mv\_prod}}$. As a help for the reader, we indicated in comment brackets the processors concerned.

Composed modules The parallel frame program for a node representing an input node of a composed module is similar to a module call, the program contains design decision information and a program fragment for block begin, i.e., an opening curly bracket, is generated. This program fragment is concatenated without any operator in front of the program fragment representing the block inside the composed module. A parallel frame program for a node representing an output node of a composed module is simply a program fragment for block end, i.e., a closing curly bracket. This program fragment is concatenated without any operator behind the program fragment representing the block inside the composed module. Example: in Figure 28 lines 01 and 02 contain the program fragment for the composed module and block begin, respectively. The program fragment for block end is not indicated in this example.

Loops/Conditionals The parallel frame program fragment for a loop/conditional is similar. For a loop/conditional input node we generate the program code for loop/conditional begin, i.e., $\mathtt{L_{t_i}}(\ldots)$ or $\mathtt{I_{t_i}}(\ldots)$, respectively , and for block begin, behind this we concatenate the code for the inner part of the loop/conditional, and at the end the code for a block end. Note that the program code for loop/conditional begin contains information about the processors assigned to modules inside the loop/conditional. Example: in Figure 28, a sequential loop can be found in line 03, the block begin opening curly bracket in line 04; the corresponding closing curly bracket is not indicated in this example.

Blocks The construction process conserves block structures specified by begin/end or input/output nodes. In order to obtain a correct order for program fragments for block begin, inner part of a block, and block end, we generate apart these program fragments and combine them thereafter. Figure 36 details the procedure; $\mathtt{G(t_i)}$ denotes the subgraph for the structure having $\mathtt{t_i}$ as begin or input node, the $\cdot$ operator used denotes program concatenation, i.e., simple string concatenation without additional operator symbol between operands.

```
while (node $t_i \in$ G not yet processed)
   if ($t_i$ is start node) // of a block, loop, etc.
      specify corresponding end node $t'_i$;
      select subgraph G($t_i$);
      select subgraph G'($t_i$)=G($t_i$)\{$t_i$,$t'_i$};
      P($t_i$)=construct_program ($t_i$);
      P(G'($t_i$))=construct_program (G'($t_i$));
      P($t'_i$)=construct_program ($t'_i$);
      P($t_i$)=P($t_i$)·P(G'($t_i$))·P($t'_i$);
      connect P($t_i$) to already available program;
   else ...
```

Figure 36: Transformation program for block structures.

Edges A scheduling graph edge (t_i, t_j) represents a sequential order of the corresponding module activations, i.e., the module activation corresponding to t_i has to be completed before the module activation corresponding to t_j can be started. The program

$$\{P(t_i)\} \circ \{P(t_j)\}$$

fulfills the requirements for execution order of program fragments. Brackets are only added if necessary. If t_i is an input or t_j is an output node, then program fragments are concatenated without operator and without brackets:

$$P(t_i)\, P(t_j)$$

If there are several edges $(t_i, t_{j_1}), \ldots, (t_i, t_{j_n})$ with a common source and no other edges between the nodes $t_{j_1}, \ldots, t_{j_n}$ (Figure 37 gives an example), then the program fragments $P(t_{j_1}), \ldots, P(t_{j_n})$ are independent and can be performed in parallel. The following program fragment is generated:

$$\{P(t_i)\} \circ \{P(t_{j_1}) \parallel \ldots \parallel P(t_{j_n})\}$$

Constructed program fragment
P(3)
$\circ$ { P(4) $\parallel$ P(5) $\parallel$ P(6) }

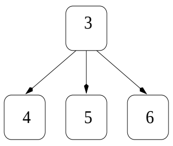


Figure 37: Scheduling graph (fork structure).

The following lemmas make statements concerning the structure of the resulting parallel frame program going out from the structure of the scheduling graph.

The first one, Lemma 3, proves that the program fragments are in the same order like the corresponding nodes in the scheduling graph. A path $(t_i, \ldots, t_j)$

in the scheduling graph signals that the corresponding tasks have to be performed sequentially. The lemma proves that the corresponding program fragments respect this order. Lemma 4 proves that disjoint paths with common source and end nodes produce parallel program fragments executed concurrently. The situations are illustrated in Figure 38.

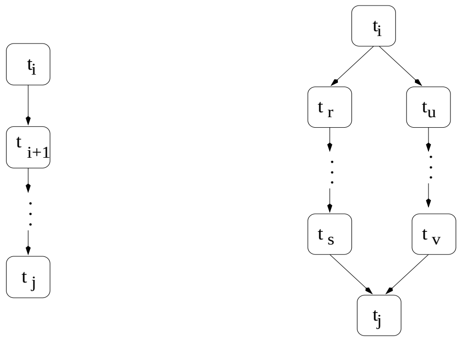


Figure 38: Scheduling graphs illustrating Lemma 3 (left) and Lemma 4.

Lemma 3 (Structure of the parallel frame program (1))
Let $(\mathtt{t_i}, \ldots, \mathtt{t_j})$ *be a path in the scheduling graph, let* $\mathtt{P(t_i)}, \ldots, \mathtt{P(t_j)}$ *be the program fragments corresponding to the nodes* $\mathtt{t_i}, \ldots, \mathtt{t_j}$*, and let* $\mathtt{P_{t_i \to t_j}}$ *be the parallel frame program fragment corresponding to the path* $(\mathtt{t_i}, \ldots, \mathtt{t_j})$ *generated by means of the construction steps described above.*
Then, the program fragments $\mathtt{P(t_i)}, \ldots, \mathtt{P(t_j)}$ *occur in* $\mathtt{P_{t_i \to t_j}}$ *in the same order as the corresponding elements* $\mathtt{t_i}, \ldots, \mathtt{t_j}$ *within the path. If we omit superfluous brackets, then the program* $\mathtt{P_{t_i \to t_j}}$ *has the structure* $\mathtt{P_1} \circ \mathtt{P_2} \circ \ldots \circ \mathtt{P_n}$*,* $1 \leq \mathtt{n} \leq \mathtt{j} - \mathtt{i}$*, the program fragments* $\mathtt{P_2}, \ldots, \mathtt{P_{n-1}}$ *have a regular block structure, the program fragments* $\mathtt{P_1}$*,* $\mathtt{P_n}$ *can be incomplete blocks if* $(\mathtt{t_i}, \ldots, \mathtt{t_j})$ *contains solely the input or solely the output node of a block.*

Proof: We demonstrate the lemma by induction on $\mathtt{k} = \mathtt{j} - \mathtt{i}$, the length of the path.

k=1: Let $(\mathtt{t_i}, \mathtt{t_j})$ be an edge in the scheduling graph. We consider the possibilities for the corresponding program fragments. Each one of the nodes $\mathtt{t_i}$ and $\mathtt{t_j}$ may correspond to input, output, or module nodes. It is not necessary to distinguish between the various input node types. In the following let $\mathtt{X}$, $\mathtt{X'}$ be one of the strings $\mathtt{M_{t_i}}(\ldots)$, $\mathtt{for}(\ldots)$, $\mathtt{if}(\ldots)$, or another begin of a block or loop structure. The following combinations can occur:

- $\mathtt{t_i}$ and $\mathtt{t_j}$ correspond to modules. Then $\mathtt{P_{t_i \to t_j}} \equiv \{\mathtt{P(t_i)}\} \circ \{\mathtt{P(t_j)}\}$
- $\mathtt{t_i}$ corresponds to a module and $\mathtt{t_j}$ corresponds to an input node. Then $\mathtt{P_{t_i \to t_j}} \equiv \{\mathtt{P(t_i)}\} \circ \mathtt{X}\{$, one complete and one incomplete block.
- $\mathtt{t_i}$ corresponds to a module and $\mathtt{t_j}$ corresponds to an output node. Then $\mathtt{P_{t_i \to t_j}} \equiv \mathtt{P(t_i)}\}$, an incomplete block.
- $\mathtt{t_i}$ corresponds to an input node and $\mathtt{t_j}$ corresponds to a module. Then $\mathtt{P_{t_i \to t_j}} \equiv \mathtt{X}\{\mathtt{P(t_j)}$, an incomplete block.

- $\mathtt{t_i}$ and $\mathtt{t_j}$ correspond to input nodes. Then $\mathtt{P_{t_i \to t_j}} \equiv \mathtt{X}\{\mathtt{X'}\{$, an incomplete block (the included block is uninteresting).

- $\mathtt{t_i}$ corresponds to an input node and $\mathtt{t_j}$ corresponds to an output node. Then $\mathtt{P_{t_i \to t_j}} \equiv \mathtt{X}\{\}$, an (empty) block.

- $\mathtt{t_i}$ corresponds to an output node and $\mathtt{t_j}$ corresponds to a module. Then $\mathtt{P_{t_i \to t_j}} \equiv \} \circ \{\mathtt{P(t_j)}\}$, one complete and one incomplete block.

- $\mathtt{t_i}$ corresponds to an output node and $\mathtt{t_j}$ corresponds to an input node. Then $\mathtt{P_{t_i \to t_j}} \equiv \} \circ \mathtt{X}\{$, two incomplete blocks.

- $\mathtt{t_i}$ corresponds to a module and $\mathtt{t_j}$ corresponds to an output node. Then $\mathtt{P_{t_i \to t_j}} \equiv \mathtt{P(t_i)}\}$, an incomplete block.

$\mathtt{k} \to \mathtt{k}+\mathtt{1}$: The induction step is similar to the first step.

□

Lemma 4 (Structure of the parallel frame program (2))
Let $(\mathtt{t_i}, \mathtt{t_r}, \ldots, \mathtt{t_s}, \mathtt{t_j})$ *and* $(\mathtt{t_i}, \mathtt{t_u}, \ldots, \mathtt{t_v}, \mathtt{t_j})$*, be paths in the scheduling graph with* $(\mathtt{t_r}, \ldots, \mathtt{t_s})$ *and* $(\mathtt{t_u}, \ldots, \mathtt{t_v})$ *disjoint subpaths, and let* $\mathtt{P_{t_i \to t_j}}$ *be the corresponding parallel frame program generated by means of the construction steps described above.*
Then the program $\mathtt{P_{t_i \to t_j}}$ *has the structure* $\mathtt{P_1} \circ \{\mathtt{P_{t_r \to t_s}} \parallel \mathtt{P_{t_u \to t_v}}\} \circ \mathtt{P_3}$*, where* $\mathtt{P_1}$, $\mathtt{P_3}$ *can be incomplete blocks, if* $(\mathtt{t_i}, \ldots, \mathtt{t_j})$ *contains solely the input or solely the output node of a block.*

Proof: For an illustration confer Figure 38 right. We demonstrate the lemma by induction on $\mathtt{k} = \mathtt{j} - \mathtt{i}$, the length of the path.

k=2: let $(\mathtt{t_i}, \mathtt{t_u}, \mathtt{t_j})$ and $(\mathtt{t_i}, \mathtt{t_r}, \mathtt{t_j})$ be paths with length 2 in the scheduling graph, $\mathtt{t_u} \neq \mathtt{t_r}$. Like in Lemma 3, we consider the possibilities for the corresponding program fragments. Each one of the nodes $\mathtt{t_i}$ and $\mathtt{t_j}$ can correspond to input, output, or module nodes. $\mathtt{t_r}$ and $\mathtt{t_u}$ solely correspond to modules, they cannot be input/output nodes because it is not possible to bridge input/output nodes. In the following let $\mathtt{X}$, $\mathtt{X'}$ be one of the strings $\mathtt{M_{t_i}}(\ldots)$, or $\mathtt{for}(\ldots)$, or another string generated for a block begin.

- $\mathtt{t_i}$ and $\mathtt{t_j}$ correspond to modules. Then $\mathtt{P_{t_i \to t_j}} \equiv \{\mathtt{P(t_i)}\} \circ \{\mathtt{P(t_u)} \parallel \mathtt{P(t_r)}\} \circ \{\mathtt{P(t_j)}\}$

- $\mathtt{t_i}$ corresponds to a module and $\mathtt{t_j}$ corresponds to an input node. Then $\mathtt{P_{t_i \to t_j}} \equiv \{\mathtt{P(t_i)}\} \circ \{\mathtt{P(t_u)} \parallel \mathtt{P(t_r)}\} \circ \mathtt{X}\{$. The last block is incomplete.

- $\mathtt{t_i}$ corresponds to a module and $\mathtt{t_j}$ corresponds to an output node. Then $\mathtt{P_{t_i \to t_j}} \equiv \{\mathtt{P(t_i)}\} \circ \{\mathtt{P(t_u)} \parallel \mathtt{P(t_r)}\}\}$, one incomplete block.

- $\mathtt{t_i}$ corresponds to an input node and $\mathtt{t_j}$ corresponds to a module. Then $\mathtt{P_{t_i \to t_j}} \equiv \mathtt{X}\{\{\mathtt{P(t_u)} \parallel \mathtt{P(t_r)}\} \circ \{\mathtt{P(t_j)}\}$, an incomplete block.

- $\mathtt{t_i}$ and $\mathtt{t_j}$ correspond to input nodes. Then $\mathtt{P_{t_i \to t_j}} \equiv \mathtt{X}\{\{\mathtt{P(t_u)} \parallel \mathtt{P(t_r)}\} \circ \mathtt{X'}\{$, an incomplete block.

- $\mathtt{t_i}$ corresponds to an input node and $\mathtt{t_j}$ corresponds to an output node. Then $\mathtt{P_{t_i \to t_j}} \equiv \mathtt{X}\{\{\mathtt{P(t_u)} \parallel \mathtt{P(t_r)}\}\}$, a block.

- $\mathtt{t_i}$ corresponds to an output node and $\mathtt{t_j}$ corresponds to a module. Then $\mathtt{P_{t_i \to t_j}} \equiv \} \circ \{\mathtt{P(t_u)} \parallel \mathtt{P(t_r)}\} \circ \{\mathtt{P(t_j)}\}$. The first block is incomplete.

- $\mathtt{t_i}$ corresponds to an output node and $\mathtt{t_j}$ corresponds to an input node. Then $\mathtt{P_{t_i \to t_j}} \equiv \} \circ \{\mathtt{P(t_u)} \parallel \mathtt{P(t_r)}\} \circ \mathtt{X}\{$. The first block and the last block are incomplete.

- $\mathtt{t_i}$ corresponds to a module and $\mathtt{t_j}$ corresponds to an output node. Then $\mathtt{P_{t_i \to t_j}} \equiv \{\mathtt{P(t_i)}\} \circ \{\mathtt{P(t_u)} \parallel \mathtt{P(t_r)}\}\}$, an incomplete block.

$\mathtt{k} \to \mathtt{k+1}$: The induction step is similar to the first step.

□

10.4 Additional Examples

Beside the examples already given, we want to illustrate the management of some typical scheduling graph structures. See in Figures 39, 41, 43, 42, 40, and 44 some typical joint, mesh, tree, and chain structures.

```
Constructed program fragment
 { P(4) || P(5) || P(6) }
 ○ P(7)
```

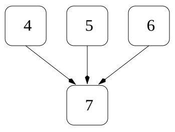


Figure 39: Scheduling graph (joint structure).

```
Constructed program fragment
 P(3)
 ○ {{ P(4) ○ {P(6) || P(7) }}
     || { P(5) ○ {P(8) || P(9) }}
   }
```

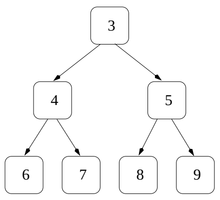


Figure 40: Scheduling graph (tree structure).

```
Constructed program fragment
 {{{ P(3)
      ○{{P(4) ○ P(6) }
          || { P(5) ○ P(8) }
    } }
    ○ P(7)
  }
  ○ { P(9) || P(10) }
 }
 ○ P(11)
```

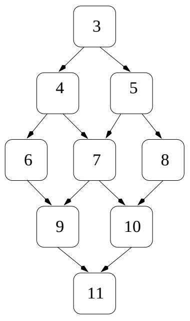


Figure 41: Scheduling graph (mesh structure).

```
Constructed program fragment
 { P(3)
   ∘ {{{ P(4) ∘ { P(6) ‖ P(7)}
          }
         ∘ P(10)}
       ‖ {{ P(5) ∘ { P(8) ‖ P(9)}}
            ∘ P(11)}}
 }
∘ P(12)
```

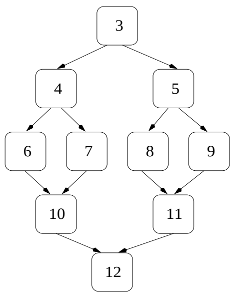


Figure 42: Scheduling graph (tree-mesh structure).

```
Constructed program fragment
 { P(3)
   ∘ { P(6)
        ‖ {{ P(4)
              ∘ { P(7)
                   ‖ {{ P(5)
                          ∘ { P(8) ‖ P(9)}
                         }
                         ∘ P(10)}}
              }
              ∘ P(11)}}
 }
 ∘ P(12)
```

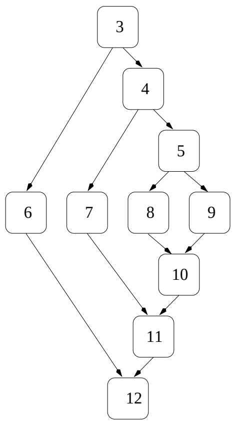


Figure 43: Scheduling graph (large mesh structure).

```
Constructed program fragment
 { { { { { P(3)
           ∘ { P(4) ‖ P(5)}
          }
          ∘ P(6)
         }
         ∘ { P(7) ‖ P(8)}
       }
       ∘ P(9)
    }
    ∘ { P(10) ‖ P(11)}
 }
 ∘ P(12)
```

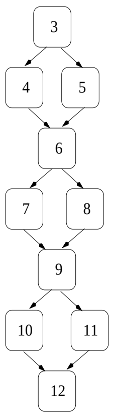


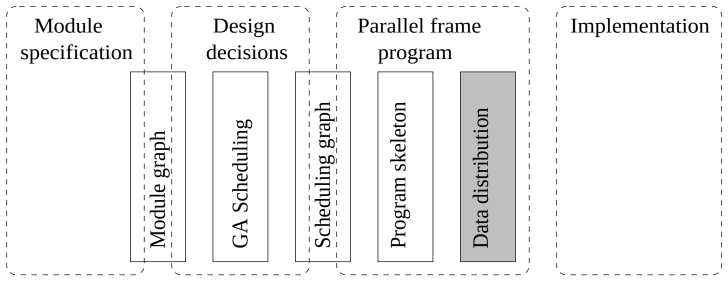
Figure 44: Scheduling graph (chain structure).

10.5 Section's Summary and Outlook

In this section we generated the skeleton of the parallel frame program which meets the scheduling decisions taken in the sections before. The construction process works on the scheduling graph. The process is recursive, a graph is recursively divided in subgraphs for which we construct program fragments. This program fragments are than connected together to a program. In order to describe this process, we specified the procedure for selecting subgraphs (Section 10.1) and we established the rules for passing through subgraphs (Section 10.2). Passing through a subgraph is a process which is often interrupted and resumed, so we specified the rules for resuming the process, too. Additionally, we optimized for a particular case the program constructed.

In Section 10.3 we specified the program parts constructed for atomic scheduling graph elements, i.e., for nodes and edges, as well as the rules for connecting constructed program parts together. In two lemmas we proved statements about the structure of the program constructed. The section ends with some additional examples (Section 10.4) which illustrate some typical scheduling graphs and the corresponding parallel frame program fragments.

This program skeleton contains all significant information concerning control sequences, execution order of tasks, and assignment of processor groups, but it lacks all data and processor management information. This information will be added to the parallel frame program in the next section. We decided to divide the construction of the parallel frame program into several steps, in order to obtain a more readable (in the sense, more readable for the user) program where the information is "as local as possible". For example, we generate the communication environment necessary for a communication just before the communication has to be done and not at the beginning of the program, although the information is already available at the beginning.



11 Data Distribution

In the section before we specified the execution order of tasks, but we did not make any assumption about data distribution. In this section we describe in detail the process of adding data distribution modules. First, we will insert distribution modules, subsequently, in a separate step (cf. Section 11.2), we will establish the communication environment necessary.

11.1 Inserting Data Redistribution Modules

The skeleton of the parallel frame program was constructed by means of the scheduling graph which was specified only for this purpose. The scheduling graph stores only a part of the data edges which represent the data flow between module activations, i.e., it stores only data edges which are indispensable for a correct relative order of the module activations. In this section we need the entire data flow information, thus we work on the initial module graph constructed in Section 5; all referred edges are *module graph edges*. In the following, we complete Table 1 by outlining the data distribution modules inserted.

11.1.1 Modules

Figure 45 illustrates the program fragment constructed for module nodes. At a node $\mathtt{t_i}$ of the scheduling graph, each variable $\mathtt{x}$ has a well defined data distribution $\mathtt{d^x_{t_i}}$. This distribution was selected by the design decision process from a set of possible data distributions. In the parallel frame program, the module call $\mathtt{M_{t_i}}(\ldots)$ of the module activation corresponding to node $\mathtt{t_i}$ lists the variables together with their data distributions and, additionally, it specifies $\mathtt{c_{M_{t_i}}}$, the communicator including all processors of $\mathtt{p_i}$, the group of processors used for executing the module call.

Data distribution has to be inserted in order to ensure a correct data flow between two module activations which use different data distributions. In terms of our data structures, data distribution accrues when two nodes of the *module graph* are connected via a data edge and the distribution of the corresponding variable, stored in the *scheduling graph*, differs in the concerned modules. In the following we indicate the rules for adding data distribution modules to the parallel frame program.

```
• $t_i$ module node
  [ $(t_h, t_i)$ module graph data edge corresponding to $x_1, \dots, x_k$ ]
  [ $(t_i, t_j)$ module graph data edge corresponding to $y_1, \dots, y_l$ ]

Recv ($x_1 : d_{t_h}^{x_1} : \mathbf{proc}\ p_h,\ x_1 : d_{t_i}^{x_1} : \mathbf{proc}\ p_i,\ c_{h,i}[p_h \cup p_i]$)
∘ ...
∘ Recv ($x_k : d_{t_h}^{x_k} : \mathbf{proc}\ p_h,\ x_k : d_{t_i}^{x_k} : \mathbf{proc}\ p_i,\ c_{h,i}[p_h \cup p_i]$)
∘ Create_domain ($\mathbf{proc}\ p_i,\ c_{M_{t_i}}$)
∘ $M_{t_i}$ ([in] $x_1 : d_{t_i}^{x_1}, \dots, x_k : d_{t_i}^{x_k}$, [out] $y_1 : d_{t_i}^{y_1}, \dots, y_l : d_{t_i}^{y_l},\ c_{M_{t_i}}[p_i]$)
∘ Free_domain ($c_{M_{t_i}}$)
∘ Send ($y_1 : d_{t_i}^{y_1} : \mathbf{proc}\ p_i,\ y_1 : d_{t_i}^{y_1} : \mathbf{proc}\ p_j,\ c_{i,j}[p_i \cup p_j]$)
∘ ...
∘ Send ($y_l : d_{t_i}^{y_l} : \mathbf{proc}\ p_i,\ y_l : d_{t_i}^{y_l} : \mathbf{proc}\ p_j,\ c_{i,j}[p_i \cup p_j]$)
```

Figure 45: Parallel frame program fragments with data distribution modules for module nodes. Brackets [...] include comments.

```
module M(in a:vec double(3,200), out c:vec double(3,200), comm $c_M$)
{ Send(a:(3,200):proc(1,2,3), a:(2,300):proc(1,2), $c_{M,A}$)
  ∘ Recv(a:(3,200):proc(1,2,3), a:(2,300):proc(1,2), $c_{M,A}$)
  ∘ Create_domain (proc (1,2), $c_A$)
  ∘ A([in]a:(2,300),[out]b:(2,300), c:(2,1), $c_A$)
  ∘ Free_domain ($c_A$)
  ∘ Send(b:(2,300):proc(1,2),b:(2,1):proc(1,2), $c_{A,B}$)
  ∘ Send(b:(2,300):proc(1,2),b:(1,600):proc(3), $c_{A,C}$)
  ∘ Send(c:(2,1):proc(1,2),c:(1,600):proc(3), $c_{A,C}$)
  ∘ { { Recv(b:(2,300):proc(1,2),b:(2,1):proc(1,2), $c_{A,B}$)
        ∘ Create_domain (proc (1,2), $c_B$)
        ∘ B([in]b:(2,1), $c_B$[1,2])
        ∘ Free_domain ($c_B$)
      }
      || { Recv(b:(2,300):proc(1,2),b:(1,600):proc(3), $c_{A,C}$)
         ∘ Recv(c:(2,1):proc(1,2),c:(1,600):proc(3), $c_{A,C}$)
         ∘ C([in]b:(1,600), c:(1,600),[out]c:(1,600), proc(3))
         ∘ Send(c:(1,600):proc(3),c:(3,200):proc(1,2,3), $c_{C,M}$)
         }
    }
  ∘ Recv(c:(1,600):proc(3),c:(3,200):proc(1,2,3), $c_{C,M}$)
}
```

Figure 46: Parallel frame program sequence with data distribution modules. Processors used are 1, 2, and 3. The corresponding module specification is `module M{A ∘ {B || C}}`. Brackets [...] include comments.

Data distribution modules A data edge (t_i, t_j) of the module graph represents a data flow from node t_i to node t_j. Let x be a variable which corresponds to this data edge, i.e., x is output variable of t_i and input variable of t_j. If the data distributions $\mathtt{d}^{\mathtt{x}}_{\mathtt{t_i}}$ and $\mathtt{d}^{\mathtt{x}}_{\mathtt{t_j}}$ are not identical, then a data distribution module for x has to be inserted. For a data distribution operation, there are two, usually distinct, processor groups involved: processor group $\mathtt{p_i}$ realizes the send and processors group $\mathtt{p_j}$ realizes the receive part of the data transfer, respectively. The processors of these two groups communicate via a well defined *common communicator* denoted as $\mathtt{c_{i,j}}$ (cf. Section 11.2 for details on establishing the communicators). Thus, in order to distribute variable x, we insert two modules, a send and a receive module. The send module (parameter's in/outs are indicated in comment brackets)

$$\mathtt{Send}\ ([\mathrm{in}]\mathtt{x} : \mathtt{d}^{\mathtt{x}}_{\mathtt{t_i}} : \textbf{proc}\ \mathtt{p_i},\ [\mathrm{out}]\ \mathtt{x} : \mathtt{d}^{\mathtt{x}}_{\mathtt{t_j}} : \textbf{proc}\ \mathtt{p_j},\ \mathtt{c_{i,j}})$$

is inserted after $\mathtt{P(t_i)}$, the program fragment corresponding to node $\mathtt{t_i}$. The receive module

$$\mathtt{Recv}\ ([\mathrm{in}]\mathtt{x} : \mathtt{d}^{\mathtt{x}}_{\mathtt{t_i}} : \textbf{proc}\ \mathtt{p_i},\ [\mathrm{out}]\ \mathtt{x} : \mathtt{d}^{\mathtt{x}}_{\mathtt{t_j}} : \textbf{proc}\ \mathtt{p_j},\ \mathtt{c_{i,j}})$$

is inserted before $\mathtt{P(t_j)}$, the program fragment corresponding to node $\mathtt{t_j}$.
$\mathtt{Send}\ (\mathtt{x} : \mathtt{d}^{\mathtt{x}}_{\mathtt{t_i}}, \ldots)$ and $\mathtt{Recv}\ (\mathtt{x} : \mathtt{d}^{\mathtt{x}}_{\mathtt{t_i}}, \ldots)$ denote empty strings if node $\mathtt{t_i}$ has no ingoing or outgoing data edges annotated with x, respectively. In this case the corresponding connecting $\circ$ operator is omitted, too.

Note 1 The module calls Send and Recv used for data distribution denote data transfer operations performed between groups of processors. When generating the corresponding program by using message passing libraries like, e.g., MPI, these distribution operations have to be implemented by several point to point communication operations.

Note 2 An output variable y of a module $\mathtt{t_i}$ can be used by several different modules $\mathtt{t_{j_1}}$, ..., $\mathtt{t_{j_k}}$ as input, thus several Send module calls have to be generated, although in Figure 45 we indicated always only one such module per variable. These modules are executed sequentially:

$$\begin{aligned}
&\mathtt{Send}\ (\mathtt{y} : \mathtt{d}^{\mathtt{y}}_{\mathtt{t_i}} : \textbf{proc}\,\mathtt{p_i},\ \mathtt{y} : \mathtt{d}^{\mathtt{y}}_{\mathtt{t_{j_1}}} : \textbf{proc}\,\mathtt{p_{j_1}},\ \mathtt{c_{i,j_1}})\\
&\circ\,\mathtt{Send}\ (\mathtt{y} : \mathtt{d}^{\mathtt{y}}_{\mathtt{t_i}} : \textbf{proc}\,\mathtt{p_i},\ \mathtt{y} : \mathtt{d}^{\mathtt{y}}_{\mathtt{t_{j_2}}} : \textbf{proc}\,\mathtt{p_{j_2}},\ \mathtt{c_{i,j_2}})\\
&\circ\ \ldots\\
&\circ\,\mathtt{Send}\ (\mathtt{y} : \mathtt{d}^{\mathtt{y}}_{\mathtt{t_i}} : \textbf{proc}\,\mathtt{p_i},\ \mathtt{y} : \mathtt{d}^{\mathtt{y}}_{\mathtt{t_{j_k}}} : \textbf{proc}\,\mathtt{p_{j_k}},\ \mathtt{c_{i,j_k}})
\end{aligned}$$

Figure 46 illustrates a program fragment with data distribution modules.

11.1.2 Composed Modules

Input and output nodes for composed modules are similar to nodes corresponding to basic modules, but concerning the data flow, input and output variables of the composed module reverse roles. Figure 47 illustrates the parallel program fragments constructed for a composed module. A module graph data edge $\mathtt{(C_0,t_j)}$ represents a data flow from an *input* variable of the composed module to an *input* variable of the inner module and data has to be redistributed according to the data distribution $\mathtt{d}^{\mathtt{x}}_{\mathtt{t_j}}$ required by the inner module. Data edges $\mathtt{(t_h,C_1)}$ for *output* variables of the composed module came from *output* variables of inner modules and have to be redistributed in order to become the distribution required by the parameter list of the composed module. In Figure 46, the very first send-receive pair (with communicator $\mathtt{c_{M,A}}$) and the very last send-receive pair (with communicator $\mathtt{c_{C,M}}$) represent such data distributions.

```
• $t_i$ input node of a composed module, $t_i \equiv C_0$
  [ $(t_i, t_j)$ module graph data edge corresponding to $x_1, \dots, x_k$ ]

$M_{t_i}$ ([in] $x_1 : d_{t_i}^{x_1}, \dots, x_k : d_{t_i}^{x_k}$, [out] $y_1 : d_{t_i}^{y_1}, \dots, y_l : d_{t_i}^{y_l}$, $c_{M_{t_i}}[p_i]$)
  {
   Send ($x_1 : d_{t_i}^{x_1}$ : **proc** $p_i$, $x_1 : d_{t_j}^{x_1}$ : **proc** $p_j$, $c_{i,j}[p_i \cup p_j]$)
   ∘ ...
   ∘ Send ($x_k : d_{t_i}^{x_k}$ : **proc** $p_i$, $x_k : d_{t_j}^{x_k}$ : **proc** $p_j$, $c_{i,j}[p_i \cup p_j]$)

• $t_i$ output node of a composed module, $t_i \equiv C_1$
  [ $(t_h, t_i)$ module graph data edge corresponding to $y_1, \dots, y_l$ ]

   Recv ($y_1 : d_{t_h}^{y_1}$ : **proc** $p_h$, $y_1 : d_{t_i}^{y_1}$ : **proc** $p_i$, $c_{h,i}[p_h \cup p_i]$)
   ∘ ...
   ∘ Recv ($y_l : d_{t_h}^{y_l}$ : **proc** $p_h$, $y_l : d_{t_i}^{y_l}$ : **proc** $p_i$, $c_{h,i}[p_h \cup p_i]$)
  }
```

Figure 47: Parallel frame program fragments with data distribution modules for composed module nodes. Brackets [. . .] include comments.

```
• $t_i$ input node of a sequential for loop, $t_i \equiv F_0$
   [ $(t_h, t_i)$ data edge of the module graph, $t_h$ outside the loop
     $A_{x_i}$, $i = 1, \dots, k$ nodes inside the loop ]

 Recv ($x_1 : d_{t_h}^{x_1}$ : **proc** $p_h$, $x_1 : d_{A_{x_1}}^{x_1}$ : **proc** $p_{A_{x_1}}$, $c_{h,A_{x_1}}[p_h \cup p_{A_{x_1}}]$)
 ∘ ...
 ∘ Recv ($x_k : d_{t_h}^{x_k}$ : **proc** $p_h$, $x_k : d_{A_{x_k}}^{x_k}$ : **proc** $p_{A_{x_k}}$, $c_{h,A_{x_k}}[p_h \cup p_{A_{x_k}}]$)
 ∘ Create_domain (**proc** $p_i$, $c_F$)
 ∘ for($i = 0, \dots, n$, $c_F[p_i]$)
    {

• $t_i$ output node of a sequential for loop, $t_i \equiv F_1$
   [ $(t_h, t_i)$ data edge of the module graph, $t_h$ inside the loop
     $(t_i, t_j)$ data edge of the module graph, $t_j$ outside the loop
     $A_{x_i}$, $i = 1, \dots, k$, $B_{y_i}$, $i = 1, \dots, l$ nodes inside the loop ]

     Recv ($x_1$: $d_{t_h}^{x_1}$: **proc** $p_h$, $x_1$: $d_{A_{x_1}}^{x_1}$ : **proc** $p_{A_{x_1}}$, $c_{h,A_{x_1}}[p_h \cup p_{A_{x_1}}]$)
     ∘ ...
     ∘ Recv ($x_k$: $d_{t_h}^{x_k}$: **proc** $p_h$, $x_k$: $d_{A_{x_k}}^{x_k}$ : **proc** $p_{A_{x_k}}$, $c_{h,A_{x_k}}[p_h \cup p_{A_{x_k}}]$)
    }
    ∘ Free_domain ($c_F[p_i]$)
    ∘ Send ($y_1 : d_{B_{y_1}}^{y_1}$ : **proc** $p_{B_{y_1}}$, $y_1 : d_{t_j}^{y_1}$ : **proc** $p_j$, $c_{B_{y_1},j}[p_{B_{y_1}} \cup p_j]$)
    ∘ ...
    ∘ Send ($y_l : d_{B_{y_l}}^{y_l}$ : **proc** $p_{B_{y_l}}$, $y_l : d_{t_j}^{y_l}$ : **proc** $p_j$, $c_{B_{y_l},j}[p_{B_{y_l}} \cup p_j]$)
```

Figure 48: Parallel frame program fragments with data distribution modules for sequential loop nodes. Brackets [. . .] include comments.

11.1.3 Sequential Loops

The parallel frame program fragments constructed for a sequential loop are illustrated in Figure 48. Input and output nodes of the loops have no own parameters, these nodes gather in their parameter lists parameter information from inside the loop (cf. Section 5.3.2 for an overview on data edges occurring in a subgraph for sequential loops). When a send or a receive operation is generated for a parameter of such a node, then the corresponding information, i.e., actual data distribution and group of processors involved, is taken from the corresponding node located inside the loop. In Figure 48 let $\mathtt{x}$ be a parameter overtaken from an inner node $\mathtt{A_x}$, $\mathtt{d^x_{A_x}}$ denotes the actual data distribution of parameter $\mathtt{x}$ in $\mathtt{A_x}$, and $\mathtt{p_{A_x}}$ denotes the group of processors mapped to the module activation $\mathtt{A_x}$.

```
(01)         { Create_domain(communicator for P)
(02)          ∘ P(...)
(03)          ∘ Free_domain(communicator for P)
(04)-(07)     ∘ Send(data generated by P)
(08)         }
(09)-(13)    ∘{ Recv(data generated outside the loop and used in the first iterat.)
(14)            ∘ Create_domain(communicator for for loop)
(15)            ∘ for(...)
(16)              {{{{ Create_domain(communicator for A)
(17)                   ∘ A(...)
(18)                   ∘ Free_domain(communicator for A)
(19)                   ∘ Send(data generated by A and used by subsequent
                              modules in the same iteration)
(20)                   ∘ Send(data generated by A and used in the next iteration)
(21)                  }∘{Recv(data used by B and generated by previous modules
                              in the same iteration )
(22)                     ∘ Create_domain(communicator for B)
(23)                     ∘ B(...)
(24)                     ∘ Free_domain(communicator for B)
(25)-(26)                ∘ Send(data generated by B and used in the next iterat.)
(27)                    }
(28)                 }
(29)                 || C(...)
(30)                }
(31)-(33)           ∘ Recv(data used in the next iteration)
(34)              } // end for-loop
(35)            ∘ Free_domain(communicator for for loop)
(36)-(39)       ∘ Send(data generated by the loop and used outside the loop)
(40)           }
(41)          ∘{ Recv(data used by Q)
(42)             ∘ Create_domain(communicator for Q)
(43)             ∘ Q(...)
(44)             ∘ Free_domain(communicator for Q)
(45)            }
```

Figure 49: Overview on a parallel frame program fragment with data distribution modules. For the module specification cf. Figure 9. Line numbers refer to the detailed program in Figure 50.

```
{ Create_domain (proc(1,2,3), c_P)
  ∘ P( ..., [out] a:d_P, b:d_P, c_P[1,2,3])
  ∘ Free_domain (c_P)
  ∘ Send( a:d_P:proc(1,2,3), a:d_A:proc(1,2), c_{P,A}[1,2,3])
  ∘ Send( b:d_P:proc(1,2,3), b:d_A:proc(1,2), c_{P,A}[1,2,3])
  ∘ Send( a:d_P:proc(1,2,3), a:d_B:proc(1,2), c_{P,B}[1,2,3])
  ∘ Send( b:d_P:proc(1,2,3), b:d_C:proc(3), c_{P,C}[1,2,3])
}
∘{ Recv( a:d_P:proc(1,2,3), a:d_A:proc(1,2), c_{P,A}[1,2,3])
   ∘ Recv( b:d_P:proc(1,2,3), b:d_A:proc(1,2), c_{P,A}[1,2,3])
   ∘ Recv( a:d_P:proc(1,2,3), a:d_B:proc(1,2), c_{P,B}[1,2,3])
   ∘ Recv( b:d_P:proc(1,2,3), b:d_C:proc(3), c_{P,C}[1,2,3])
   ∘ Recv( d:d_*:proc(*), c:d_C:proc(3), c_{*,C}[*,3])
   ∘ Create_domain (proc(1,2,3), c_F)
   ∘ for (..., c_F[1,2,3])
      {{{{Create_domain (proc(1,2), c_A)
             ∘ A([in] a:d_A b:d_A, [out] c:d_A, d:d_A, c_A[1,2])
             ∘ Free_domain (c_A)
             ∘ Send( c:d_A:proc(1,2), c:d_B:proc(1,2), c_{A,B}[1,2])
             ∘ Send( d:d_A:proc(1,2), d:d_C:proc(3), c_{A,C}[1,2,3])
           }∘{Recv( c:d_A:proc(1,2), c:d_B:proc(1,2), c_{A,B}[1,2])
               ∘ Create_domain (proc(1,2), c_B)
               ∘ B([in] a:d_B c:d_B, [out] b:d_B, c_B[1,2])
               ∘ Free_domain (c_B)
               ∘ Send(b:d_B:proc(1,2), b:d_A:proc(1,2), c_{B,A}[1,2])
               ∘ Send(b:d_B:proc(1,2), b:d_C:proc(3), c_{B,C}[1,2,3])
             }
         }
         || C([in] b:d_C, d:d_C, [out] e:d_C, c_C[3])
        }
        ∘ Recv( b:d_B:proc(1,2), b:d_A:proc(1,2), c_{B,A}[1,2])
        ∘ Recv( b:d_B:proc(1,2), b:d_C:proc(3), c_{B,C}[1,2,3])
        ∘ Recv( d:d_A:proc(1,2), d:d_C:proc(3), c_{A,C}[1,2,3])
      }
   ∘ Free_domain (c_F)
   ∘ Send( b:d_B:proc(1,2), b:d_*:proc(*), c_{B,*}[1,2,*])
   ∘ Send( c:d_A:proc(1,2), c:d_*:proc(*), c_{A,*}[1,2,*])
   ∘ Send( d:d_A:proc(1,2), d:d_*:proc(*), c_{A,*}[1,2,*])
   ∘ Send( e:d_C:proc(3), e:d_Q:proc(1,2,3), c_{C,Q}[1,2,3])
  }
  ∘{ Recv( e:d_C:proc(3), e:d_Q:proc(1,2,3), c_{C,Q}[1,2,3])
     ∘ Create_domain (proc(1,2,3), c_Q)
     ∘ Q([in] e:d_Q, ..., c_Q[1,2,3])
     ∘ Free_domain (c_Q)
   }
```

Figure 50: Parallel frame program fragment. '*' denotes modules or module information coming from outside the program fragment illustrated.

Example Figure 9 illustrates the module graph for a sequential loop; as a general rule we can say that `Send` and `Recv` module calls have to be constructed for all data edges illustrated in this figure. Figures 49 and 50 illustrate a therefore constructed parallel frame program fragment. For data edges incoming to the input node of the sequential loop and which represent data generated outside the loop and used inside the loop, especially in the first iteration step, `Recv` module calls are generated (lines 09-13). For data edges outgoing from the output node of the loop and which represent data generated inside the loop and used by modules outside the loop, program code for `Send` module calls is generated (lines 36-39). These modules are performed only once, before beginning the loop, and after end of the loop, respectively. Accordingly, these module calls have to be placed outside the loop. Data edges incoming to the output node of the loop represent data distribution operations for data generated in the loop and used in the next iteration by modules inside the loop. These data operations have to be performed each time when iterating the loop and, thus, have to be placed inside the loop (`Recv` operations in lines 31-33). The corresponding `Send` calls are inside the loop and are placed as usual after the module activations which generate the corresponding data (lines 20 and 25-26, respectively).

11.1.4 Parallel Loops

```
• $t_i$ input node of a parallel parfor loop, $t_i \equiv P_0$
  [ $(t_h, t_i)$ data edge of the module graph,
    $A_{x_i}$, $i = 1, \ldots, k$ nodes inside the loop ]

Recv ($x_1 : d^{x_1}_{t_h}$ : **proc** $p_h$, $x_1 : d^{x_1}_{A_{x_1}}$ : **proc** $p_{A_{x_1}}$, $c_{h,A_{x_1}}[p_h \cup p_{A_{x_1}}]$)
○ ...
○ Recv ($x_k : d^{x_k}_{t_h}$ : **proc** $p_h$, $x_k : d^{x_k}_{A_{x_k}}$ : **proc** $p_{A_{x_k}}$, $c_{h,A_{x_k}}[p_h \cup p_{A_{x_k}}]$)
○ Create_domain (**proc** $p_i$, $c_P$)
○ parfor ($c_P[p_i]$)
   {

• $t_i$ output node of a parallel parfor loop, $t_i \equiv P_1$
  [ $(t_i, t_j)$ data edge of the module graph,
    $B_{y_i}$, $i = 1, \ldots, l$ nodes inside the loop ]

   }
   ○ Free_domain ($c_P[p_i]$)
   ○ Send ($y_1 : d^{y_1}_{B_{y_1}}$ : **proc** $p_{B_{y_1}}$, $y_1 : d^{y_1}_{t_j}$ : **proc** $p_j$, $c_{B_{y_1},j}[p_{B_{y_1}} \cup p_j]$)
   ○ ...
   ○ Send ($y_1 : d^{y_1}_{B_{y_1}}$ : **proc** $p_{B_{y_1}}$, $y_1 : d^{y_1}_{t_j}$ : **proc** $p_j$, $c_{B_{y_1},j}[p_{B_{y_1}} \cup p_j]$)
```

Figure 51: Parallel frame program with data distribution modules for parallel loops.

Figure 51 illustrates the parallel frame program fragment constructed for parallel loops. The principle is the same as before: we generate data distribution module calls for each module graph data edge (an example of a module graph with a parallel loop can be found in Figure 10).

11.1.5 Conditionals

Figure 52 illustrates the parallel frame program fragment constructed for a conditional, the corresponding module graph can be found in Figure 11.

```
• $t_i$ input node of a conditional, $t_i \equiv I_0$
  [ $(t_h, t_i)$ data edge of the module graph,
    $A_{x_i}$, $i = 1, \dots, k$ nodes inside the conditional ]

Recv ($x_1 : d^{x_1}_{t_h}$ : proc $p_h$, $x_1 : d^{x_1}_{A_{x_1}}$ : proc $p_{A_{x_1}}$, $c_{h,A_{x_1}}[p_h \cup p_{A_{x_1}}]$)
∘ ...
∘ Recv ($x_k : d^{x_k}_{t_h}$ : proc $p_h$, $x_k : d^{x_k}_{A_{x_k}}$ : proc $p_{A_{x_k}}$, $c_{h,A_{x_k}}[p_h \cup p_{A_{x_k}}]$)
∘ Create_domain (proc $p_i$, $c_I$)
∘ if(..., $c_I[p_i]$)
   {

• $t_i$ output node of a conditional, $t_i \equiv I_1$
  [ $(t_i, t_j)$ data edge of the module graph,
    $B_{y_i}$, $i = 1, \dots, l$ nodes inside the conditional ]

   }
   ∘ Free_domain ($c_I[p_i]$)
   ∘ Send ($y_1 : d^{y_1}_{B_{y_1}}$ : proc $p_{B_{y_1}}$, $y_1 : d^{y_1}_{t_j}$ : proc $p_j$, $c_{B_{y_1},j}[p_{B_{y_1}} \cup p_j]$)
   ∘ ...
   ∘ Send ($y_1 : d^{y_1}_{B_{y_1}}$ : proc $p_{B_{y_1}}$, $y_1 : d^{y_1}_{t_j}$ : proc $p_j$, $c_{B_{y_1},j}[p_{B_{y_1}} \cup p_j]$)
```

Figure 52: Parallel frame program with data distribution modules for conditionals.

11.2 Establishing Communicators

Any point to point communication occurs within a well defined domain, the communication domain. We use the notion of communicator in the sense defined by MPI [65], as being the local representation of the global communication domain. Other message passing libraries have similar notions. For message passing libraries which do not support groups, collective communication on subsets of the processors have to be simulated in more troublesome way.

In order to describe how to establish communicators, first, in Section 11.2.1, we describe the data structure used, then, in Section 11.2.2 we establish the communicators for the point to point communication domains.

11.2.1 Abstract Syntax Trees

In order to establish the communicators, we use abstract syntax trees. An *abstract syntax tree* is a tree where each node is an operator and the son nodes represent the corresponding operands. Our abstract syntax trees represent parallel frame programs. Therefore we use $\circ$ and $||$ operands, and, additionally, we treat the composed module itself, loops, and conditionals like operators: the particular operator node is labeled by `comp_mod`, `for`, `parfor`, or `if`, the left son points to the abstract syntax tree or to the subtree corresponding to the inner block of the loop or conditional, the right son is empty.

In order to use abstract syntax trees for representing *parallel* programs, we have to take into consideration that we use several processors or groups of processors and that processors can compute different program fragments. The nodes of the abstract syntax tree have annotations which specify the set of processors involved in the computation of the corresponding node or of the subtree having this node as root. The semantics is as follows: for a specified processor `p`, we consider only the nodes which have `p` in the annotations. It is obvious that the abstract syntax trees corresponding to the operations performed by each one of the processors are subtrees of the entire abstract syntax tree.

In the next sections, we establish communicators for various data distributions. The cases which have to be considered depend on the node's type, i.e., module node, input/output node of a loop or a conditional, or composed module node, and they depend on where the nodes are located, i.e., inside or outside a loop or conditional.

11.2.2 Communicators for Module to Module Communication

All tasks within this section correspond to module activations. Within the activation of send and receive modules we indicated in Section 11.1 the communicators for realizing the communication, but up to now we did not introduce any operation for establishing the communication domain. The reason is that when passing through the scheduling graph, the (global) information needed for establishing the communication domains is locally not available. Figure 53 illustrates this. Here, data transfer has to be done between task 3 and 6, and 4 and 8, respectively. In Figure 53 data transfer is sketched by dashed lines, these edges do not belong to the scheduling graph. In order to do, e.g., data transfer from task 4 to task 8, a communication domain including processors 0, 1, and 2 has to be established (numbers in paranthesis represent processors mapped to tasks). When processing node 4, which use solely processor 1, we do not have any information about the state of processors 0 and 2, and, additionally, we do not known, if processors 0 and 2 passes through this program point.

Corresponding program fragment

```
P(2)
∘ { { { P(3) || P(4) }
        ∘ P(6)
    }
    || { P(5) ∘ P(7) }
  }
∘ P(8)
```

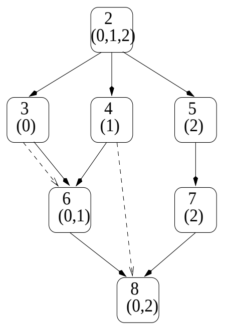


Figure 53: Scheduling graph with additional data distribution information.

Corresponding program fragment

```
P(2)
∘ { { { P(3) || P(4) }
        ∘y P(6)
      }
      || { P(5) ∘ P(7) }
   }
∘x P(8)
```

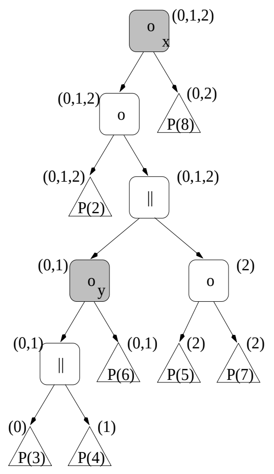


Figure 54: Abstract syntax tree.

Let us construct the parallel program fragment for this scheduling subgraph and lookup particularly to the corresponding abstract syntax tree illustrated in Figure 54. The `x` marked $\circ$ operator node is a common program point for the processor from task 4 and the processors from task 8, all these processors pass this program point, i.e., all these processors are in the node's set of assigned processors. The x-node is the first common predecessor of `P(4)` and `P(8)` in the abstract syntax tree. Thus, we will establish the common communication domain for the program fragments `P(4)` and `P(8)` in the vicinity of this program point. Analogously, the `y` marked node is the common synchronization point for the processor from task 3 and the processors from task 6.

Synchronization point A synchronization point for a set of processors which execute a program is a program point which is passed by all processors of the set, i.e., a synchronization point is a node of the abstract syntax tree which includes in its set of assigned processors all processors concerned.

In general, for a data communication represented in the module graph by a data edge $(\mathtt{t_i},\mathtt{t_j})$, we have to find the already existing synchronization point for the processor groups $\mathtt{p_i}$ and $\mathtt{p_j}$, without adding new synchronization operations. For example, $\circ$ operator nodes inside the corresponding parallel frame program represent natural synchronization points of the program. Thus, we have to find the operator node which connects the two program fragments $\mathtt{P_i}$ and $\mathtt{P_j}$, containing $\mathtt{P(t_i)}$ and $\mathtt{P(t_j)}$, respectively. This operator node exists, it is the first common predecessor of $\mathtt{P(t_i)}$ and $\mathtt{P(t_j)}$ (see Lemma 5 below). After finding this node, we insert the program fragment for initiating the communication domain so that the communicator is established before entering the program fragment $\mathtt{P_i}$ and we insert the program fragment for closing the communication domain after ending $\mathtt{P_j}$. Therefore, the structure of the program will be:

$$
\begin{array}{l}
\mathtt{Create\_domain}\,(\mathbf{proc}\;\mathtt{p_i} \cup \mathtt{p_j},\ \mathtt{c_{i,j}}) \\
\circ\ \{\ \mathtt{P_i}\ \circ\ \mathtt{P_j}\ \} \\
\circ\ \mathtt{Free\_domain}\,(\mathtt{c_{i,j}}).
\end{array}
$$

Corresponding program fragment

```
{Create_domain((0, 1, 2), c4,8)
 ∘ {{P(2)
     ∘ {{{Create_domain((0, 1), c3,6)
          ∘ {{P(3) ‖ P(4)}
              ∘y P(6)
            }
         }
        ∘ Free_domain(c3,6)
       }
       ‖  {P(5) ∘ P(7)}
    }
   }
   ∘x P(8)
  }
}
∘ Free_domain(c4,8)
```

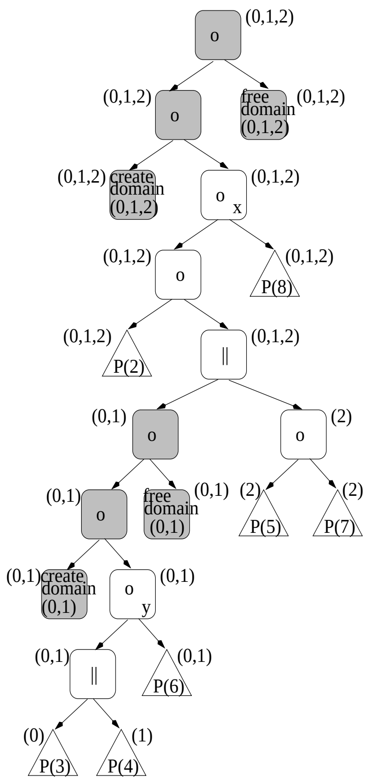


Figure 55: Abstract syntax tree with communication domain operations.

Example In Figure 55 we illustrate the result of this procedure for the abstract syntax tree from Figure 54. Added nodes are shaded. In order to easier distinguish the operators in the program fragment, we indexed with `x/y` the $\circ$ operators corresponding to the nodes marked. The modules `Create_domain` and `Free_domain` create/free the communication domain consisting on the processors indicated. For example, the domain for the communication between nodes 4 and 8 is created in the very first line and freed in the last program line. For readability reasons, the corresponding `Send` and `Recv` communication operations are omitted.

Overview on the algorithm Figure 56 gives an overview on the algorithm used for finding the synchronization points and adding the domain management operations for module to module data transfer. Only the case that $\mathtt{t_i}$, $\mathtt{t_j}$ are both module nodes located on the same level, i.e., both outside a loop or conditional, or both inside a loop or `if` statement, is considered. The right figure illustrates the abstract syntax tree used within the algorithm for the case that the concerned node `x` is a left son. For this algorithm we use the abstract syntax tree (cf. Section 11.2.1) of the parallel frame program generated in Section 10. Note that the algorithm works on the abstract syntax tree, but it also uses additional information stored in the module graph, e.g., information about variables to which data edges correspond.

```
∀ data edge $(t_i, t_j) \in$ module graph
and $t_i$, $t_j$ modules outside a loop
if( $\exists$ variable $v \in$ annotation $(t_i, t_j)$
        with $d^v_{t_i} \neq d^v_{t_j}$ )
   // henceforth the algorithm works
   // on the abstract syntax tree
 { x   = find_common_pred(P($t_i$), P($t_j$));
   cn  = new node(create_node, $p_i \cup p_j$);
   fn  = new node(free_node, $p_i \cup p_j$);
   $o_1$  = new node($\circ$_node, $p_x$);
   $o_2$  = new node($\circ$_node, $p_x$);
   fx  = father(x);
   $o_1$.l_son = cn;
   $o_1$.r_son = x;
   $o_2$.l_son = $o_1$;
   $o_2$.r_son = fn;
   if (fx.r_son == x) fx.r_son = $o_2$;
   else fx.l_son = $o_2$;
 }
```

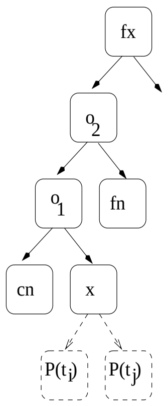


Figure 56: Algorithm to add communication domain management operations for module to module communication.

The next lemma proves the existence of synchronization points for all possible node types participating at a data transfer operation. Subsequently, communication domain operations will be inserted in the vicinity of this point (see below).

Lemma 5 (Synchronization points)
Let $(\mathtt{t_i}, \mathtt{t_j})$ *be a data edge of the module graph, let* `ast` *be the abstract syntax tree generated by means of the algorithm described in Section 10, and let* $\mathtt{p_i}$, $\mathtt{p_j}$ *be the processor groups assigned to the module activations concerned.*

The synchronization point for the processors in $\mathtt{p_i}$ *and* $\mathtt{p_j}$ *is either an* `ast` *operator node labeled with* $\circ$*, or the root node of a* `for`*,* `parfor`*, or* `if` *labeled subtree of* `ast`*, or the root node of* `ast`*.*

Proof: The cases which have to be considered depend on the nodes' type, i.e., module node, input/output node of a loop or a conditional, or composed module node, and depend on where the nodes are located, i.e., inside or outside a loop or conditional.

Case 1: Let $\mathtt{t_i}$, $\mathtt{t_j}$ be nodes corresponding to module activations, and let $\mathtt{t_i}$, $\mathtt{t_j}$ be both outside a loop or conditional, or let both be inside a loop or conditional. According to Lemma 1 about scheduling graph information, for a data edge $(\mathtt{t_i}, \mathtt{t_j})$ of the module graph there is a path $(\mathtt{t_i}, \ldots, \mathtt{t_j})$ in the scheduling graph. According to Lemma 3 and 4 about the structure of the program, the corresponding parallel frame program is $\mathtt{P_1} \circ \mathtt{P_2} \circ \ldots \circ \mathtt{P_n}$, i.e., the corresponding part of `ast` has the form illustrated in Figure 57.

On their part, the program fragments $\mathtt{P_l}$, $\mathtt{l} = \mathtt{1}, \ldots, \mathtt{n}$, have (cf. proof of Lemma 3 and 4) the form $\mathtt{X}\{\ldots\mathtt{P(t_k)}\ldots\}$ with $\mathtt{X}$ empty or $\mathtt{X}$ = `for`(),

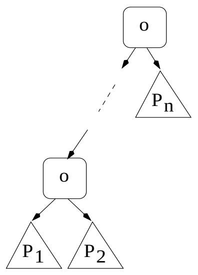


Figure 57: Abstract syntax tree for $\mathtt{P_1} \circ \mathtt{P_2} \circ \ldots \circ \mathtt{P_n}$.

$\mathtt{X} = \mathtt{parfor}()$, or $\mathtt{X} = \mathtt{if}()$ for $\mathtt{k} = \mathtt{i}, \ldots, \mathtt{j}$. Particularly $\mathtt{P_1} = \mathtt{X}\{\ldots \mathtt{P(t_i)} \ldots\}$ and $\mathtt{P_n} = \mathtt{X}\{\ldots \mathtt{P(t_j)} \ldots\}$. Thus, $\mathtt{p_i}$ and $\mathtt{p_j}$ are subgroups of the processor group assigned to the root of `ast`. It is obvious that all processors assigned to parts of `ast` have to pass the root, an operator node labeled with $\circ$. Additionally, the root of `ast` is the first common node for these processors.

Case 2: Let one of the nodes $\mathtt{t_i}$, $\mathtt{t_j}$ be corresponding to a module activation outside a loop or outside a conditional and the other node be corresponding to the input/output node of a loop or a conditional.
In the corresponding parallel program $\mathtt{P_1} \circ \mathtt{P_2} \circ \ldots \circ \mathtt{P_n}$, the $\mathtt{P_1}$ or the $\mathtt{P_n}$ part, i.e., the part corresponding to the input/output node, is an incomplete block (cf. proof of Lemma 3 and 4), but the proof can be done similarly to the first case above.

Case 3: Let one of the nodes $\mathtt{t_i}$, $\mathtt{t_j}$ be corresponding to a module activation inside a loop or conditional and the other node be corresponding to the input/output node of the loop or conditional. Looked at more closely (cf. Section 5), the only possible situation is that $\mathtt{t_i}$ is inside a sequential loop and $\mathtt{t_j}$ is the output node of the loop. Data edge $(\mathtt{t_i}, \mathtt{t_j})$ represents data operations caused by sequential looping (cf. Section 5.3.2) and which have to be performed for each loop iteration. The loop's root node, i.e., the `for` labeled `ast` node, is synchronization point for the corresponding processors, all processors assigned to the loop are elements of the `for` node's processor set.

Case 4: Let one of the nodes $\mathtt{t_i}$, $\mathtt{t_j}$ be corresponding to an input or output node of the composed module. The data edges concerned represent data operations for the composed module's input or output data. The `ast`'s root node, i.e., the `comp_mod` labeled `ast` node, is synchronization point for the corresponding processors, all processors assigned to the composed module are elements of the root's processor set.

$\square$

11.2.3 Communicators for Block Input/Output Nodes

Let $(\mathtt{t_i}, \mathtt{t_j})$ be a data edge and let one of the nodes be corresponding to the input/output node of a loop or a conditional. As shown in the proof of Lemma 5, the case that the other node corresponds to a module activation outside the loop or outside the conditional can be treated similarly to the module to module case seen in Section 11.2.2.

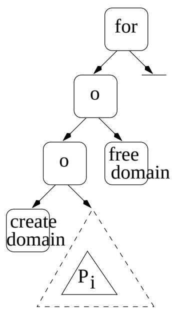


Figure 58: Algorithm to add communication domain management operations for module to `for` output node communication.

In the lemma's proof, we also showed that the case that one of the nodes $\mathtt{t_i}$, $\mathtt{t_j}$ is corresponding to a module activation inside a loop or conditional and the other node is corresponding to the input/output node of the loop or conditional, can be reduced to: $\mathtt{t_i}$ is inside a sequential loop and $\mathtt{t_j}$ is the output node of the loop. Figure 58 illustrates the algorithm to add communication domain management operations.

11.2.4 Communicators for Composed Modules

Let $(\mathtt{t_i}, \mathtt{t_j})$ be a data edge where one of the nodes $\mathtt{t_i}$, $\mathtt{t_j}$ corresponds to an input or output node of the composed module. The data edges concerned represent data operations for the composed module's input or output data. The algorithm to add communication domain management operations is similar to the case above (Section 11.2.3).

11.3 Section's Summary and Outlook

In this generation step, first we added data distribution modules to the program skeleton constructed in the sections before. Therefore, we described in detail the construction steps for adding data distribution modules to the program fragments constructed for the basic language elements, i.e., for modules (Section 11.1.1), composed modules (Section 11.1.2), loops (Section 11.1.3 and 11.1.4), and conditionals (Section 11.1.5).

Subsequently, we added communication environment management procedures to the until then constructed parallel frame program. Therefore, we used the abstract syntax tree of the program constructed in the steps before (Section 11.2.1), and we added the corresponding operations by means of this syntax tree (Sections 11.2.2, 11.2.3, and 11.2.4). In a lemma we demonstrated the correctness of our construction steps, i.e., we proved the existence of program points which are passed by the corresponding processors and, around these points, the algorithm adds communication domain management operations. The next section presents the complete parallel frame program for the conjugate gradient method.

Now, the parallel frame program contains all information necessary for the implementation of a parallel program for a specified target machine. The final C program with MPI calls can be obtained by a simple transformation of this parallel frame program. We renounce to describe this syntax driven transformation step.

12 Example

Parallel frame program for the conjugate gradient iteration with 4 processors, i.e., processors 0, 1, 2, and 3, and an assumed vector size of 300.

```
module CG (in A:mat double(4,75,1,1), p:vec double(4,75),
        p_0:vec double(4,75), w:vec double(4,75), w_0:vec double(4,75),
        r:vec double(4,75), r_0:vec double(4,75), x:vec double(4,75),
        λ_0:double:proc(3), out p:vec double(4,75), x:vec double(4,75),
        r:vec double(4,75), comm c_CG[0,1,2,3])
{ Create_domain (proc (0,1,2,3), c_{CG,for})
  ∘ Send (w_0:(4,75):proc(0,1,2,3), w_0:(1,300):proc(3), c_{CG,for})
  ∘ Send (r_0:(4,75):proc(0,1,2,3), r_0:(1,300):proc(0), c_{CG,for})
  ∘ Send (p:(4,75):proc(0,1,2,3), p:(1,300):proc(0), c_{CG,for})
  ∘ Recv (w_0:(4,75):proc(0,1,2,3), w_0:(1,300):proc(3), c_{CG,for})
  ∘ Recv (r_0:(4,75):proc(0,1,2,3), r_0:(1,300):proc(0), c_{CG,for})
  ∘ Recv (p:(4,75):proc(0,1,2,3), p:(1,300):proc(0), c_{CG,for})
  ∘ Create_domain (proc (0,1,2,3), c_for)
  ∘ for (k=0..K, c_for)
   {
     ∘ Create_domain (proc (0,1,2,3), c_{vv_sub_2,for})
     ∘ Create_domain (proc (0,1,2,3), c_{mv_prod,vv_prod_2})
     ∘ Create_domain (proc (0,1,2,3), c_{mv_prod,vv_prod_3})
     ∘ Create_domain (proc (0,1,2,3), c_{mv_prod,vv_prod_4})
     ∘ Create_domain (proc (0,1,2,3), c_{mv_prod})
     ∘ mv_prod ([in]A:mat(4,75,1,1), p:(4,75),[out]w:(4,75), c_{mv_prod})
     ∘ Free_domain (c_{mv_prod})
     ∘ Send (w:(4,75):proc(0,1,2,3),w:(1,300):proc(1), c_{mv_prod,vv_prod_2})
     ∘ Send (w:(4,75):proc(0,1,2,3),w:(1,300):proc(2), c_{mv_prod,vv_prod_3})
     ∘ Send (w:(4,75):proc(0,1,2,3),w:(1,300):proc(3), c_{mv_prod,vv_prod_4})
     ∘ Create_domain (proc (0,1,2,3), c_{vv_prod_1,op_assign_1})
     ∘ Create_domain (proc (0,1,2,3), c_{vv_prod_2,op_assign_3})
     ∘ Create_domain (proc (0,1,2,3), c_{vv_prod_2,assign})
     ∘ {
         { vv_prod_1 ([in] p:(1,300), r_0:(1,300), [out]tmp_1, proc(0))
           ∘ Send ( tmp_1:proc(0), tmp_1:proc(2), c_{vv_prod_1,op_assign_1}) }
         }
         || { Recv (w:(4,75):proc(0,1,2,3), w:(1,300):proc(1),
                                                  c_{mv_prod,vv_prod_2})
              ∘ vv_prod_2 ([in]w:(1,300), p:(1,300),[out] λ_1,proc (2))
              ∘ Send (λ_1:proc(2),λ_1:proc(1), c_{vv_prod_2,op_assign_3})
              ∘ Send (λ_1:proc(2),λ_1:proc(3), c_{vv_prod_2,assign}) }
         || { Recv (w:(4,75):proc(0,1,2,3), [out]w:(1,300):proc(2),
                                                  c_{mv_prod,vv_prod_3})
              ∘ vv_prod_3 ([in]w:(1,300), w:(1,300), [out] tmp_2, proc(1))
            }
         || { Recv (w:(4,75):proc(0,1,2,3), w:(1,300):proc(3),
                                                  c_{mv_prod,vv_prod_4})
              ∘ vv_prod_4 ([in]w:(1,300), w_0:(1,300), [out] tmp_3, proc(3))
            }
       }
```

```
(40)    ∘ Free_domain ($c_{mv\_prod,vv\_prod_4}$)
(41)    ∘ Free_domain ($c_{mv\_prod,vv\_prod_3}$)
(39)    ∘ Free_domain ($c_{mv\_prod,vv\_prod_2}$)
(42)    ∘ Create_domain (proc (0,1,2,3), $c_{op\_assign_1,sv\_prod_1}$)
(43)    ∘ Create_domain (proc (0,1,2,3), $c_{op\_assign_2,sv\_prod_4}$)
(44)    ∘ Create_domain (proc (0,1,2,3),$c_{op\_assign_3,sv\_prod_3}$)
(45)    ∘ { { Recv ( $tmp_1$:proc(0), $tmp_1$:proc(2), $c_{vv\_prod_1,op\_assign_1}$)
(46)            ∘ $op\_assign_1$ ([in]$tmp_1$, op, $\lambda_1$, [out]$\xi$, proc (2))
(47)            ∘ Send ($\xi$:proc(2), $\xi$:proc(0,1,2,3), $c_{op\_assign_1,sv\_prod_1}$) }
(48)        || { $op\_assign_2$ ([in]$tmp_3$, op, $\lambda_0$, [out]$\nu$, proc (3))
(49)              ∘ Send ($\nu$:proc(3),$\nu$:proc(0,1,2,3), $c_{op\_assign_2,sv\_prod_4}$)
(50)              ∘ Recv ($\lambda_1$:proc(2),$\lambda_1$:proc(3), $c_{vv\_prod_2,assign}$)
(51)              ∘ assign ([in]$\lambda_1$,[out]$\lambda_0$, proc(3)) }
(52)        || { Recv ($\lambda_1$:proc(2),$\lambda_1$:proc(1), $c_{vv\_prod_2,op\_assign_3}$)
(53)              ∘ $op\_assign_3$ ([in]$tmp_2$, op, $\lambda_1$, [out]$\mu$, proc(1))
(54)              ∘ Send ($\mu$:proc(1),$\mu$:proc(0,1,2,3), $c_{op\_assign_3,sv\_prod_3}$) }
(55)      }
(56)    ∘ Free_domain ($c_{vv\_prod_2,assign}$)
(57)    ∘ Free_domain ($c_{vv\_prod_2,op\_assign_3}$)
(58)    ∘ Free_domain ($c_{vv\_prod_1,op\_assign_1}$)
(59)    ∘ Recv ($\xi$:proc(2),$\xi$:proc(0,1,2,3), $c_{op\_assign_1,sv\_prod_1}$)
(60)    ∘ Create_domain (proc (0,1,2,3), $c_{sv\_prod_1}$)
(61)    ∘ $sv\_prod_1$ ([in]$\xi$, p:(4,75),[out]$tmp_4$:(4,75), $c_{sv\_prod_1}$)
(62)    ∘ Free_domain ($c_{sv\_prod_1}$)
(63)    ∘ Free_domain ($c_{op\_assign_1,sv\_prod_1}$)
(64)    ∘ $vv\_assign_1$ ([in]w:(1,300), [out] $w_0$:(1,300), proc(3))
(65)    ∘ Create_domain (proc (0,1,2,3), $c_{sv\_prod_2}$)
(66)    ∘ $sv\_prod_2$ ([in]$\xi$, w:(4,75),[out]$tmp_5$:(4,75), $c_{sv\_prod_2}$)
(67)    ∘ Free_domain ($c_{sv\_prod_2}$)
(68)    ∘ Recv ($\mu$:proc(1),[out]$\mu$:proc(0,1,2,3), $c_{op\_assign_3,sv\_prod_3}$)
(69)    ∘ Create_domain (proc (0,1,2,3), $c_{sv\_prod_3}$)
(70)    ∘ $sv\_prod_3$ ([in]$\mu$, p:(4,75),[out]$tmp_6$:(4,75), $c_{sv\_prod_3}$)
(71)    ∘ Free_domain ($c_{sv\_prod_3}$)
(72)    ∘ Free_domain ($c_{op\_assign_3,sv\_prod_3}$)
(73)    ∘ Recv ($\nu$:proc(3),$\nu$:proc(0,1,2,3), $c_{op\_assign_2,sv\_prod_4}$)
(74)    ∘ Create_domain (proc (0,1,2,3), $c_{sv\_prod_4}$)
(75)    ∘ $sv\_prod_4$ ([in]$\nu$, $p_0$:(4,75),[out]$tmp_7$:(4,75), $c_{sv\_prod_4}$)
(76)    ∘ Free_domain ($c_{sv\_prod_4}$)
(77)    ∘ Free_domain ($c_{op\_assign_2,sv\_prod_4}$)
(78)    ∘ Create_domain (proc (0,1,2,3), $c_{vv\_sub_1}$)
(79)    ∘ $vv\_sub_1$ ([in]r:(4,75), $tmp_5$:(4,75),[out]r:(4,75), $c_{vv\_sub_1}$)
(80)    ∘ Free_domain ($c_{vv\_sub_1}$)
(81)    ∘ Create_domain (proc (0,1,2,3), $c_{vv\_assign_2}$)
(82)    ∘ $vv\_assign_2$ ([in]p:(4,75), [out]$p_0$:(4,75), $c_{vv\_assign_2}$)
(83)    ∘ Free_domain ($c_{vv\_assign_2}$)
(84)    ∘ Create_domain (proc (0,1,2,3), $c_{vv\_add_1}$)
(85)    ∘ $vv\_add_1$ ([in]x:(4,75), $tmp_4$:(4,75),[out]x:(4,75), $c_{vv\_add_1}$)
(86)    ∘ Free_domain ($c_{vv\_add_1}$)
(87)    ∘ Create_domain (proc (0,1,2,3), $c_{vv\_add_2}$)
(88)    ∘ $vv\_add_2$ ([in]$tmp_6$:(4,75), $tmp_7$:(4,75),[out]$tmp_6$:(4,75), $c_{vv\_add_2}$)
(89)    ∘ Free_domain ($c_{vv\_add_2}$)
```

```
    ○ Create_domain (proc (0,1,2,3), $c_{vv\_sub_2}$)
    ○ vv_sub$_2$ ([in]w:(4,75), tmp$_6$:(4,75),[out]p:(4,75), $c_{vv\_sub_2}$)
    ○ Send (p:(4,75):proc(0,1,2,3),[out]p:(1,300):proc(0), $c_{vv\_sub_2,for}$)
    ○ Free_domain ($c_{vv\_sub_2}$)
    ○ Recv (p:(4,75):proc(0,1,2,3),p:(1,300):proc(0), $c_{vv\_sub_2,for}$)
    ○ Free_domain ($c_{vv\_sub_2,for}$)
  } // end for loop
  ○ Free_domain ($c_{for}$)
  ○ Free_domain ($c_{CG,for}$)
}
```

Figure 59 illustrates the runtime for 4 processors on a Cray T3E: a straightforward, pure data parallel version and an implementation of the version presented before, which mixes task and data parallelism. This implementation is slightly better than the pure data parallel solution. Due to the low resolution and the small differences (less than 2%) between, the two graphs are overlapping in the figure. We used for our experiments as input the BCSSM matrices (number 06, 07, 09, 11, and 13) of the Harwell-Boeing Collection [44]. We denote with problem size the size of matrices used.

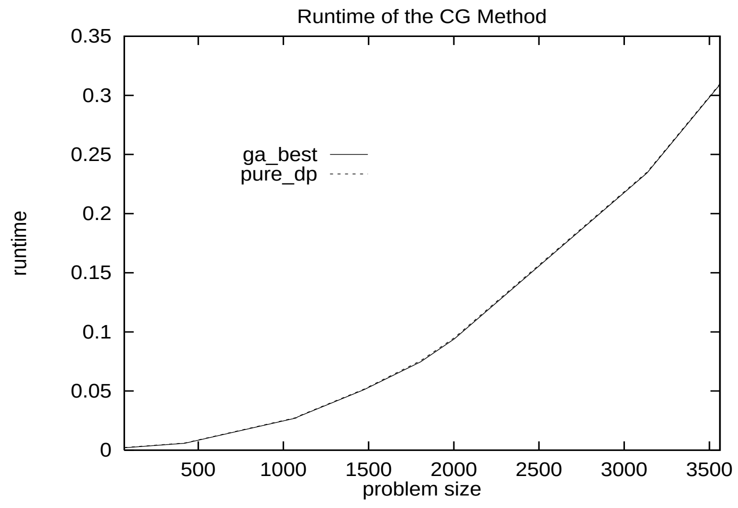


Figure 59: Runtime of the conjugate gradient iteration method for 4 processors on Cray T3E.

13 Implementation

A prototype of our tool for generating mixed task and data parallel implementations has been implemented. The implementation follows the systems' structure presented in this thesis and consists of:

- Front end; this part realizes the scanning and parsing of the user's module specification and its transformation into a module graph representation.
- Scheduling; this is the system's kernel, it realizes the genetic algorithm scheduling.
- Back end; this part realizes the interpretation of the scheduling result and its transformation into a parallel frame program and, finally, into a C+MPI program.

The *front end* is a small compiler for the module specification language. The resulting module graph which represents the users' specification contains among graph information, information about context dependence (cf. Section 5.3.3).

The *scheduling* has as input the module graph information and several files storing user provided information about available task implementation and about communication functions. To each task there is information about the implementation versions available: a specification about the number of processors necessary, information, given as distribution vector (cf. Section 6.2), about data distribution, and information about the runtime behaviour of the specified task version. The runtime information may be a measured time or it may be a link to a runtime estimation function to estimate the runtime depending, e.g., on the actual number of processors available. These runtime estimation functions are provided as program code which has to be linked to the scheduling program. Information about communication functions concern their runtime behaviour, the information provided is a code fragment implementing the specific runtime formula used. This code fragment is included into the scheduling program, i.e., the scheduling program has to be recompiled for each individual target machine. Beside these, each scheduling run is parametrized with the genetic algorithm parameters described in Section 7.7.

A scheduling run is mainly a run of the genetic algorithm. It can be started with or without initial population, i.e., the user has the possibility to include good individuals from previous runs. The number of individuals overtaken from previous runs is user determined. A scheduling run provides individuals (currently 3 individuals per run) representing the best solutions found.

The *back end* transforms individuals representing a scheduling solution into parallel frame programs which fix the scheduling decisions, i.e., schedule of tasks, processors' assignment to tasks, and for each task an implementation with a well defined data distribution for the parameters. Data distribution modules and communication domain management functions are also included into the parallel frame program. Herewith, a description of a parallel implementation which may mix task and data parallelism is provided. Finally, this program is transformed into a C program with MPI message passing interface.

The system was implemented in C/C++ on a Sun UltraSparc. The compiler for transforming module specification into module graphs uses Lex/Yacc generators, the genetic algorithm scheduling is based on the framework for generating genetic algorithms implemented by A. Thüring [68]. In order to allow an easy replacement of individual parts of the system by other modules and for test purposes, all interfaces are kept simple, so that they can be read and edited with conventional editors. Parts of the system can be replaced by other modules as long as the interfaces remain unchanged. This offer, for example, the possibility of experimenting with other schedule techniques.

Front end and back end part are not time critical, they have mainly to manage data for a single program. The scheduling is a time consuming program, due to the large search area to be inspected. A problem similar to the problem presented in Section 7.9, a genetic algorithm run with 100 individuals, 500 generations, a problem size of 3000, and a target machine with 4 processors, needs about three and a half hours on a Sun UltraSparc. In this time the genetic algorithm scheduling has found the best implementation of the conjugate gradient method, as an analysis of the possible implementations showed. In this implementation, the first 4 `vv_prod` modules are performed concurrently, the remainder of modules are performed sequentially in a data parallel manner by using a block data distribution (cf. the parallel frame program example from Section 12).

C+MPI parallel programs with mixed task and data parallelism were tested on a SPP2000 and on a Cray T3E.

14 Summary and Outlook

This thesis introduces a powerful multi-dimensional genetic algorithm based scheduling which is embedded into a tool for generating parallel programs with mixed task and data parallelism. The goal is to generate an efficient parallel implementation for an application on a distributed memory target machine, the results achieved demonstrate the efficiency of our approach.

For the efficiency of a parallel application it is essential to exploit the potential parallelism. Several applications offer at the same time a potential of task parallelism and a potential of data parallelism. We combine in our approach task and data parallelism: on the one hand we separate task and data parallelism in order to allow a complete specification of the potential parallelism of the application to be implemented, on the other hand we allow an interchange of specific information between task and data parallel level. The separation of task and data parallelism is given in the module specification where task parallelism is specified by means of the module specification language's concepts and data parallelism is hidden in basic modules implemented in a data parallel manner. We define a compact module specification language which allows the specification of task and data parallelism of an application. During the program generation process, information about module interfaces, especially about data distribution, and implementation costs information is exchanged between the levels. Information about data distribution is an attribute of each one of the several implementation versions available for modules. Data redistribution modules are added by the system, if necessary. For the implementation costs information, i.e., for estimating the consequence of design decisions, we use in our system an efficient runtime prediction formula overtaken from [57].

For the efficiency of an implementation of a parallel application it is essential to know the method to be implemented and the target machine's characteristics. One of the basic ideas of our system is to divide the work between programer and compiler, the user provides for the method specification, we provide for a compiler tool which manages the required information about the target machine's characteristics. Herewith, we assist the user in scheduling their program on a target machine and, at the same time, we eliminate for the user the need of an exact knowledge of the system architecture.

Ultimately, for the efficiency of an implementation the design decisions have the greatest influence, but the problems to be solved are complex (both, scheduling and efficient data distribution, are NP-complete [26, 43]). We propose in this thesis an efficient genetic algorithm based scheduling which takes important design decisions in order to obtain efficient implementations which simultaneously exploit task and data parallelism. The advantage of this approach is not only that it provides a scheduling algorithm, but also that it allows choosing appropriate implementations from a set of functions and convenient data distributions. The results of the scheduling step are fixed in a parallel frame program which includes all decisions taken (for each module an implementation version with a fixed number of processors for the execution and an appropriate data distribution for the parameters, the execution order of modules, and the mapping of processors to tasks) and which includes additional operations necessary for an implementation (data redistribution modules, and communication domain management operations). The parallel frame program can be translated by a syntax directed pass in any imperative language augmented by a message passing library supporting groups. For our experiments we chose C as imperative programing language and MPI as message passing interface. This syntax driven translation step is not described in this thesis.

This thesis achieved promising results in the on the area of compiler tools for the generation of parallel programs with mixed task and data parallelism. Future research is mainly directed towards investigating new approaches for parts of the system and increasing performance.

The system has a modular structure which allows parts of the system to be replaced by other modules when preserving interface and functionality. Thus, e.g., experiments with other scheduling approaches may be performed.

Our system allows in principle hierarchically structured module specification. Future research includes the exploitation of this structure for subsequent program development steps, especially scheduling for hierarchically structured module specifications. System elements were already designed for a hierarchical application.

The genetic algorithm operations' locality, i.e., operations performed during the scheduling are limited to a restricted number of individuals, indicates a parallelization potential. Thus, performing concurrently several computations and exchanging in intervals the best individuals seems to be an almost natural parallelization possibility of the scheduling itself. This is an interesting future development of our system in order to increase the computation potential and to inspect larger search spaces.

References

[1] I. Ahmad, Y.K. Kwok. On Parallelizing the Multiprocessor Scheduling Problem. In *IEEE Transactions on Parallel and Distributed Systems*, 10(4):414–432, 1999.

[2] A.V. Aho, R. Sethi, J.D.Ullman. Compilerbau. Addison-Wesley, 1995.

[3] A. Alexandrov, M. Ionescu, K.E. Schauser, C. Scheiman. LogGP: Incorporating Long Messages into the LogP model - One step closer towards a realistic model for parallel computation. University of California at Santa Barbara, TRCS95-09, 1995.

[4] F. Arbab, E. Monfroy. Using Coordination for Cooperative Constraint Solving. In *Proceedings of the 1998 ACM Symposium on Applied Computing*, pp:139–148, 1998.

[5] Arvind, L. Augustsson, J. Hicks, R. Nikhil, S.P. Jones, J. Stoy, J. Williams. pH: a parallel Haskell. http://www.abp.lcs.mit.edu, 1995.

[6] H. Bal and M. Haines. Approaches for Integrating Task and Data Parallelism. In *IEEE Concurrency*, 6(3):74–84, 1998.

[7] D. Banerjee, J.C. Browne. "Optimal" Parallelism Trough Integration of Data and Controll Parallelism: A Case Study in Complete Paralelization. In *Proceedings of the 10th International Workshop Languages and Compilers for Parallel Computing*, LNCS 1366, Springer Verlag, 1997.

[8] P. Banerjee, J. Chandy, M. Gupta, E. Hodge, J. Holmes, A. Lain, D. Palermo, S. Ramaswamy, E. Su. The Paradigm Compiler for Distributed-Memory Multicomputers. IEEE Computer Society Press, 28(10):37–47, 1995.

[9] C.J. Beckmann, C. Polychronopoulos. Microarchitecture Support for Dynamic Scheduling of Acyclic Task Graphs. University of Illinois, CSRD Report 1207, 1992.

[10] A. Bemmerl. The TOPSYS Architecture. In *Proceedings of CONPAR'90.* LNCS 457, pp:732–743, Springer Verlag, 1990.

[11] F. Berman. Experience with an Automatic Solution to the Mapping Problem. In *The Characteristics of Parallel Algorithms* ed. L.H. Jamieson, D. Gannon, R. Douglass. MIT Press, 12:307–334, 1987.

[12] D.P. Bertsekas, J.N. Tsitsiklis. Parallel and Distributed Computation. Prentice Hall, 1989.

[13] A. Bode, P. Braun. VISTOP - Visualization Tool for Parallel Systems. In *Proceedings of Visual Aspects of Man-Machine-Systems*, 1993.

[14] H. Burkhart, R. Frank, G. Hächler. ALWAN - A Coordination Language for Data Parallel Programming. In *Proceedings of the eighth SIAM Conference on Parallel Processing for Scientific Computing*, 1997.

[15] T.L. Casavant, J.G. Kuhl. A Taxonomy of Scheduling in General-Purpose Distributed Computing Systems. In *IEEE Transaction on Software Engineering*, 14(2):141–154, 1988.

[16] K.M. Chandy, J. Misra. Parallel Program Design: A Foundation. Addison-Wesley, 1988.

[17] G. Cheng, G. Fox, K. Mills. Integrating Multiple Programming Paradigms on Connection Machine CM5 in a Dataflow-Based Software Environment. Technical Report, Northeast Parallel Architecture Center, Syracuse University, 1993.

[18] C. Clémençon, K.M. Decker, A. Endo, J. Fritscher, G. Jost, N. Masuda, A. Müller, R. Rühl, W. Sawyer, E. de Sturler, B.J.N. Wylie. Application-driven Development of an Integrated Tool for Distributed Memory Parallel Processors. Technical Report CSCS-TR-94-01, CSCS-ETH, Section of Research and Development, 1994.

[19] R. Corrêa, A. Ferreira, P. Rebreyend. Scheduling Multiprocessor Tasks with Genetic Algorithms. In *IEEE Transactions on Parallel and Distributed Systems*, 10(8):825–837, 1999.

[20] D. Culler, R. Karp, A. Sahay, K. Schauser, E. Santos, R. Subramonian, T. von Eicken. LogP: Towards a realistic model of parallel computation. In *Proceedings of the 4th Symposium on Principles and Practice of Parallel Programming*, 28(4):1–12, 1993.

[21] I. De Falco, R. Del Balio, E. Tarantino. An Analysis of Parallel Heuristics for Task Allocation in Multicomputers. In *Computing: Archiv für Informatik und Numerik*, 59(3):259–275, 1997.

[22] M. Dhodhi, I. Ahmad, I. Ahmad. A Multiprocessor Scheduling Scheme Using Problem-Space Genetic Algorithms. In Proceedings of the IEEE International Conference on Evolutionary Computing, pp:214–219, 1995.

[23] A. Dierstein, R. Hayer, T. Rauber. The ADDAP System on the iPSC/860: Automatic Data Distribution and Parallelization. In *Journal of Parallel and Distributed Computing*, 32(1):1–10, 1996.

[24] M. Driscoll, W. Daasch. Accurate Predictions of Parallel Program Execution Time. In *Journal of Parallel and Distributed Computing*, 25(1):16–30, 1995.

[25] J.J. Dongarra, T. Dunigan. Message-Passing Performance of Various Computers. In *Concurrency - Practice and Experience*, 9(10):915–926, 1997.

[26] D. Fernandez-Baca. Allocating modules to processors in a distributed system. In *IEEE Transactions on Software Engineering*, SE-15, 11:1427:1436, 1989.

[27] U. Fissgus. Scheduling Using Genetic Algorithms. In *Proceedings of the 20th International Conference on Distributed Computing Systems.* IEEE Computer Society Press, pp:662-669, 2000.

[28] U. Fissgus, T. Rauber, G. Rünger. A Framework for Generating Task Parallel Programs. In *Proceedings of the 7th Symposium on the Frontiers of Massively Parallel Computation.* IEEE Computer Society Press, pp:72–80, 1999.

[29] U. Fissgus, T. Rauber, G. Rünger. A Framework for Generating Group Parallel Programs. In *Proceedings of the 7th Workshop on Compilers for Parallel Computers*, pp:105–116, 1998.

[30] I. Foster. Designing and Building Parallel Programs. Addison-Wesley, 1994.

[31] I. Foster and K.M. Chandy. Fortran M: A Language for Modular Parallel Programming. In *Journal of Parallel and Distributed Computing*, 25:1, 1995.

[32] I. Foster, M. Xu, B. Avalani, A. Choudhary. A Compilation System That Integrates High Performance Fortran and Fortran M. In *Procceedings 1994 Scalable High Performance Computing Conference*, pp:293–300, IEEE Computer Society Press, 1994.

[33] D. Goldberg. Genetic Algorithms in Search, Optimisation, and Machine Learning. Addison Wesley, 1989.

[34] L. Heinrich-Litan, U. Fissgus, St. Sutter, P. Molitor, T. Rauber. Modeling the Communication Behavior of Distributed Memory Machines by Genetic Programming. In *Proceedings of the 4th International Euro-Par Conference*, pp:273–278, 1998.

[35] High Performance Fortran Forum. High Performance Fortran Language Specification, 2.0. 1997.

[36] E.S. Hou, N. Ansari, H. Ren. A Genetic Algorithm for Multiprocessors Scheduling. In *IEEE Transactions on Parallel and Distributed Systems*, 5(2):113–120, 1994.

[37] K. Hwang, Z. Xu, M. Arakawa. Benchmark Evaluation of the IBM SP2 for Parallel Signal Processing. In *IEEE Transactions on Parallel and Distributed Systems*, 7(5):522–536, 1996.

[38] Y.K. Kwok, I. Ahmad, M.Y. Wu, W. Shu. A Graphical Tool for Automatic Paralelization and Scheduling of Programs on Multiprocessors. In *Proceedings of the third International Euro-Par Conference*, pp:294–301, 1997.

[39] E.F.A. Lederer, R.A. Dumitrescu. Two-Stage Programming. In *Proceedings of Third Fuji Interational Symposium on Functional and Logic Programming*, pp:296–311, 1998.

[40] E.F.A. Lederer, R.A. Dumitrescu. Specification-Consistent Coordination Model for Computations. In *Proceedings of 13th ACM Annual Symposium Applied Computing, Special Track on Coordination Models, Languages and Applications*, pp:122–129, 1998.

[41] T. Lewis, H. El-Rewini. Parallax: A Tool for Parallel Program Scheduling. In *IEEE Parallel & Distributed Technology*, pp:62–72, 1993.

[42] J. Li, M. Chen. Index Domain Allignment: Minimizing Costs of Cross-Referencing between Distributed Arrays. In *Proceedings of the Third Symposium on the Frontiers of Massively Parallel Computation*, pp:424–433, 1990.

[43] M. Mace. Memory Storage Patterns in Parallel Processing. Kluwer Academic, 1987.

[44] Matrix Market. http://math.nist.gov/MatrixMarket/.

[45] P. Mehrotra, M. Haines. An Overview of the Opus Langauge and Runtime System. ICASE Report 94-39. Institute for Computer Application in Science and Engineering, Hampton, VA, 1994.

[46] G. Mounié, C. Rapine, D. Trystram. Efficient Approximation Algorithms for Scheduling Malleable Tasks. In *Proceedings of the Eleventh ACM Symposium on Parallel Algorithms and Architectures*, pp:23–32, 1999.

[47] M. Norman, P. Thanisch. Models of Machines and Computation for Mapping in Multicomputers. In *ACM Computing Surveys*, 25(3):263–302, 1993.

[48] C.D. Polychronopoulos, M. Girkar, M.R. Haghighat, C.L. Lee, B. Leung, D. Schouten. Parafrase-2: An Environment for Parallelizing, Partitioning, Synchronizing, and Scheduling Programs on Multiprocessors. In *Proceedings of the 1989 International Conference on Parallel Processing.* IEEE Computer Society Press, Vol II:39–48, 1989.

[49] G. Papadopoulos. The New Dataflow Architecture being built at MIT. In *Proceedings of MIT-ZTI Symposium an Very High Parallel Architectures*, 1987.

[50] S. Ramaswamy. Simultaneous Exploitation of Task and Data Parallelism in Regular Scientific Applications. PhD thesis, University of Illinois at Urbana-Champaign, 1996.

[51] S. Ramaswamy, P. Banerjee. Automatic Generation of Efficient Array Redistribution Routines for Distributed Memory Multicomputers. In *Proceedings Frontiers '95*, pp:342–349, 1995.

[52] S. Ramaswamy, S. Sapatnekar, P. Banerjee. A Framework for Exploiting Task and Data Parallelism on Distributed-Memory Multicomputers. In *IEEE Transactions on Parallel and Distributed Systems*, 8(11):1098-1116, 1997.

[53] T. Rauber, G. Rünger. Comparing Task and Data Parallel Execution Schemes for the DIIRK method. In *Proceedings EuroPar'96*, Springer LNCS 1124, pp:52–61, 1996.

[54] T. Rauber, G. Rünger. The Compiler TwoL for the Design of Parallel Implementations. In *Proceedings 4th International Conf. on Parallel Architectures and Compilation Techniques*, IEEE, pp:282–301, 1996.

[55] T. Rauber, G. Rünger. Performance Predictions for Parallel Diagonal-Implicitely Iterated Runke-Kutta Methods. In *Proceedings of the 9th Workshop on Parallel and Distributed Simulation*, pp:21–28, 1995.

[56] T. Rauber, G. Rünger. Parallel Iterated Runke-Kutta Methods and Applications. In *International Journal of Supercomputer Applications*, 10(1):62–90, 1996.

[57] T. Rauber, G. Rünger. PVM and MPI Communication Operations on the IBM SP2: Modeling and Comparison. In *Proceedings of the HPCS'97*, pp:141–152, 1997.

[58] T. Rauber, G. Rünger. Scheduling of Multiprocessor Tasks for Numerical Applications. In *Proceedings of the 8th IEEE Symposium on Parallel and Distributed Processing*, IEEE Computer Society Press, pp:474–481, 1996.

[59] T. Rauber, G. Rünger. Scheduling of Data Parallel Modules for Scientific Computing. In *Proceedings of the 9th SIAM Conference on Parallel Processing for Scientific Computing*, SIAM CD-ROM, 1999.

[60] T. Rauber, G. Rünger. Deriving Array Distributions by Optimization Techniques. In *Journal of Supercomputing*, Vol. 15, pp:271–293, Kluwer, 2000.

[61] T. Rauber, G. Rünger. Parallele und verteilte Programmierung. Springer, 2000.

[62] H. Shachnai, J. Turek. Multiresource Malleable Task Scheduling to Minimize Response Time. In *Information Processing Letters* 70(1):47, 1999.

[63] B. Shirazi, C.K. Kavi, J. Marquis, A.R. Hurson. PARSA: A Parallel Program Software Development Tool. In *Proceedings 1994 Symposium on Assessment of Quality Software Development Tools*. IEEE Computer Society Press, 1994, pp:96–111.

[64] B.A. Shirazi, A.R. Hurson, K.M. Kavi. Scheduling and Load Balancing in Parallel and Distributed Systems. IEEE Computer Society Press, 1996.

[65] M. Snir, S.W. Otto, S. Huss-Lederman, D.W. Walker, J. Dongarra. MPI The Complete Reference. MIT Press, 1996.

[66] J. Subhlok, B. Yang. A New Model for Integrating Nested Task and Data Parallel Programming. In *Proceedings of the 8th ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, pp:1–12, 1997.

[67] J. Subhlok. Automatic Mapping of task and data Parallel programs for Efficient Execution on Multiprocessors. Technical Report CMU-CS-93-212, Carnegie Mellon University, 1993.

[68] A. Thüring. Skriptbasierte Steuerung für parallele evolutionäre Algorithmen. Diploma thesis, University Halle-Wittenberg, 1999.

[69] L.G. Valiant. A Bridging Model for Parallel Computation. In *Communications of the ACM*, 33(8):103–111, 1990.

[70] E.F. van de Velde. Concurrent Scientific Computing. Springer, 1994.

[71] L. Wang, H.J. Siegel, V.P. Roychowdhury, and A.A. Maciejewski. Task Matching and Scheduling in Heterogeneous Computing Environments Using a Genetic-Algorithm-Based Approach. *Journal of Parallel and Distributed Computing*, 47:8–22, 1997.

[72] M. Wolfe. High Performance Compilers for Parallel Computing. Addison Wesley, 1996.

Ursula Fissgus

fissgus@informatik.uni-halle.de

Persönliche Angaben

Familienstand: verheiratet, 2 Kinder (*1992,*1993)

Staatsangehörigkeit: deutsch

Geburtstag: 08.07.1957

Geburtsort: Lupeni (Rumänien)

Ausbildung

1976	**Realistisch-humanistisches Lyzeum Petroschen / Rumänien** Reifeprüfungsdiplom (Gesamtnote 1.0)
1978	**Studienkolleg der Universität des Saarlandes** Abitur (Gesamtnote 1.0)
1980	**Universität des Saarlandes** Diplomvorprüfung für Informatiker, Nebenfach Mathematik (Gesamtnote sehr gut)
1983	**Universität des Saarlandes** Diplom-Hauptprüfung für Informatiker, Nebenfach Mathematik (Gesamtnote sehr gut)
Januar 2001	**Martin-Luther-Universität Halle-Wittenberg** Einreichung der Promotionsarbeit

Beruflicher Werdegang

1980-1982

Universität des Saarlandes
Betreuung von Übungen zu den Lehrveranstaltungen
- *Systematisches Programmieren*
- *Schaltkreistheorie*
- *Algorithmentheorie*

1983-1987

Universität des Saarlandes
Wissenschaftliche Mitarbeiterin am Lehrstuhl für *Angewandte Mathematik und Theoretische Informatik* (Prof. Dr. G. Hotz) und in dem Sonderforschungsbereich 124 *VLSI Entwurfsmethoden und Parallelität* der Deutschen Forschungsgemeinschaft (DFG)

1987-1997

DIaLOGIKa Gesellschaft für Angewandte Informatik mbH
Saarbrücken, Luxembourg, Bruxelles
Projektleiterin und Senior Analyst
(08.93-01.97) Erziehungsurlaub

Nebentätigkeit während des Erziehungsurlaubes:

Fachhochschule Merseburg, FB Informatik und Angew. Naturwissenschaften
Lehrauftrag im Fachgebiet *Compilerbau*, SS 1996 und WS 1996/97

Privates Bildungsinstitut, Mannheim
Dozentin für *Objektorientierte Analyse und Design*, August 1996

1997-02.2001

Martin-Luther-Universität Halle-Wittenberg
Wissenschaftliche Mitarbeiterin im DFG-Projekt *Compilerwerkzeuge für die modulare Entwicklung von parallelen Programmen im Bereich des wissenschaftlichen Rechnens* am Lehrstuhl für *Praktische Informatik / Compilerbau* (Prof. Dr. Th. Rauber)

Nebentätigkeit:

Fachhochschule Merseburg, FB Informatik und Angew. Naturwissenschaften
Lehrauftrag im Fachgebiet *Compilerbau*, SS 1996, WS 1996/97, WS 1997/98, WS 1998/99 und WS 1999/2000

Staatliche Studienakademie Sachsen, Berufsakademie Glauchau
Lehrauftrag im Lehrgebiet *Compilerbau*, SS 1998

WS 2000/01

Fachhochschule Schmalkalden, FB Informatik
C2-Lehrstuhlvertretung (50%) *Allgemeine Informatik*

SS 2001

Fachhochschule Schmalkalden, FB Informatik
C2-Lehrstuhlvertretung (100%) *Allgemeine Informatik*

Promotionsverfahren Ursula Fissgus

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides Statt, daß ich die Arbeit selbständig und ohne fremde Hilfe verfaßt, andere als die von mir angegebenen Quellen und Hilfsmittel nicht benutzt und die den benutzten Werken wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Um einen Doktorgrad habe ich mich bisher nicht beworben.

Ursula Fissgus

Halle/Saale, den 17. Januar 2001