A Framework for Generating Task Parallel Programs

Ursula Fissgus[1] * Thomas Rauber[1] Gudula Rünger[2]

[1] Computer Science Department, University Halle–Wittenberg, 06099 Halle (Saale), Germany, {fissgus,rauber}@informatik.uni-halle.de, tel. +49-345-5524711, fax +49-345-5527009

[2] Computer Science Department, University Leipzig, Postfach 920, 04009 Leipzig, Germany, ruenger@informatik.uni-leipzig.de, tel. +49-341-9732207

Abstract

We consider the generation of mixed task and data parallel programs and discuss how a clear separation into a task and data parallel level can support the development of efficient programs. The program development starts with a specification of the maximum degree of task and data parallelism and proceeds by performing several derivation steps in which the degree of parallelism is adapted to a specific parallel machine. The separation between the task and data parallel level is preserved during the design and translation phases by clearly defined interfaces. We show how the final message-passing programs are generated from the data parallel and the task parallel specification and how the interaction between the two levels can be established. We demonstrate the usefulness of the approach by examples from numerical analysis which offer the potential of a mixed task and data parallel execution but for which it is not a priori clear, how this potential should be used for an implementation on a specific parallel machine.

Key words: task parallelism, data parallelism, parallelizing compiler, software development environment, distributed memory multiprocessors.

1 Introduction

The exploitation of different kinds of potential parallelism, e.g., task or data parallelism, is often useful for designing efficient parallel application programs but requires an integrated investigation of all kinds of potential parallelism. The detection mechanisms for potential parallelism are quite different according to their different nature, varying from automatic detection to user-guided specifications. Parallelism at operation level with a fine level of granularity is usually detected by the code generation phase of a compiler and can be 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. The granularity depends on the number of data elements per processor. Data parallelism can often be detected by parallelizing compilers using loop parallelization techniques [34]. *Task parallelism* occurs when independent program parts can be executed on different processors or disjoint groups of processors where processors of the same group collaborate in a data parallel fashion. Most applications have only a small degree of task parallelism with a coarse granularity. Nevertheless, using the available task parallelism and combining it with data parallelism can increase the performance of parallel applications considerably since an additional degree of parallelism is exploited [21, 24]. This is especially important for parallel machines with a large number of processors like the ASCI teraflop machines or many of the T3E installations. Applications that benefit from a combination of task and data parallelism include examples from

*author supported by Deutsche Forschungsgemeinschaft

numerical analysis [23, 24], signal processing [29], and multidisciplinary codes like Global Climate Modeling [2].

The integration of task and data parallelism is an active area of research because of its potential benefits and several approaches have been proposed recently. These include language approaches like Fortran M, Fx, and Opus which add task parallel constructs to High Performance Fortran and compiler approaches like Paradigm aiming at an automatic detection of task and data parallel computations [21]. Data parallel constructs as an extension to task parallel languages have also been proposed like in Orca and Braid, see [2] for a good overview and comparison.

In this article, we describe a general framework for generating programs with mixed task parallel and data parallel executions on distributed-memory machines (DMMs) based on the TwoL model (Two-Level parallelism). Previous papers on this programming model concentrate on the question whether the exploitation of the available degree of task parallelism of a specific application leads to an efficient program for a given DMM and how design decisions like task scheduling and data distribution can be derived from a general specification expressing the potential parallelism [25, 26]. In this paper, we consider the relationship between task and data parallel executions and describe how hierarchically structured task parallelism can automatically be transformed into a corresponding message-passing program using a hierarchy of group contexts. Tasks that are not further decomposed are executed in a data parallel way. The generation of parallel programs with mixed task and data parallelism is embedded into the framework which defines clear interfaces between task and data parallel parts. The framework can be used to develop parallel programs in a systematic way by clearly separating task and data parallel parts, but it also describes the structure of a compiler system for the derivation of parallel programs from a high-level specification. The framework is flexible enough to allow interactive, semiautomatic, or automatic support for the design and realization of efficient parallel programs.

In the following, we describe the framework in more detail and show how the final program is generated. In particular, we show how the high-level specification of task parallelism is translated into MPI programs that have the correct interface to the data parallel level. The actual connection to the data parallel level can be established in different ways. If a library of data parallel realizations of basic computations is available, e.g., in the form of a scientific library like ScaLAPACK [5], correct function calls are created by the translation step of the task parallel part, i.e., function calls with the data distribution required by the library interface. The specification of the processor groups that execute the data parallel parts can be provided by the programmer by annotations. If such data parallel realizations are not available, a similar design step as in the task level from a specification to a parallel program can be provided.

The rest of the paper is organized as follows. Section 2 describes the framework with a conjugate gradient method as example. Section 3 describes the task parallel part and the translation into a message-passing program. Section 4 sketches a specification mechanism of the data parallel part. Section 5 shows how the framework can be applied to several examples from numerical analysis to select the most efficient implementations. Section 6 compares the proposed framework to similar approaches in the area of parallelizing compilers and Section 7 concludes.

2 Model Overview

In this section, we present our model for integrating task and data parallelism and apply it to an example from numerical analysis.

2.1 Separation of task and data parallelism

The TwoL model clearly separates task parallelism from data parallelism. Data parallelism is expressed by basic modules (BMs) which can be provided as executable parallel programs operating on arbitrary data or in form of a high-level data parallel specification. Task parallelism is expressed by the programmer in form of a *module specification* which consists of a set of composed modules.

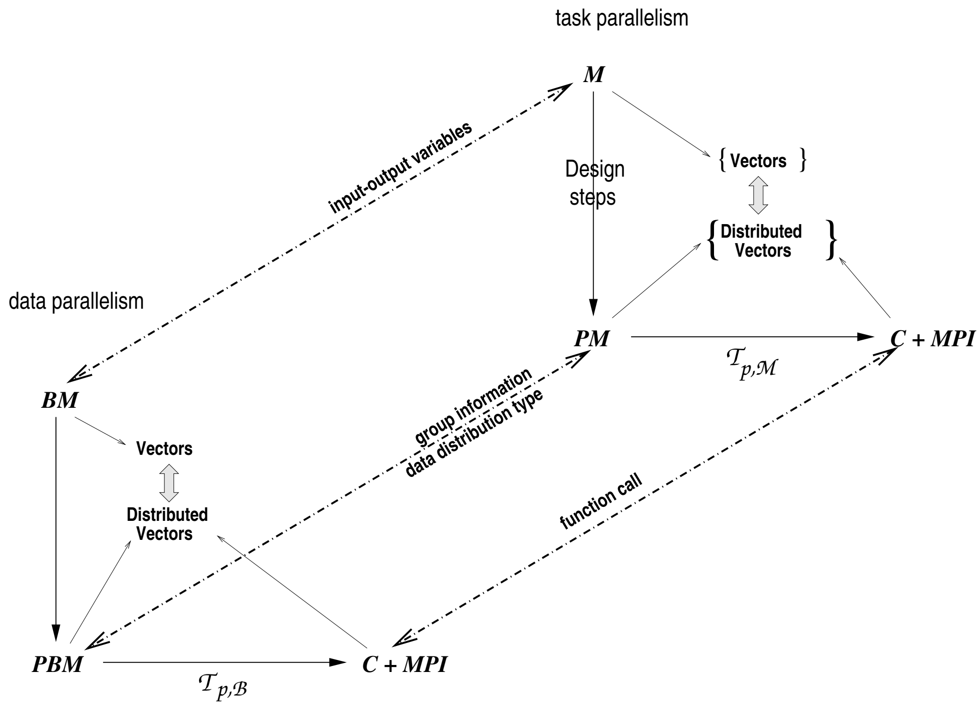

Figure 1: Overview of translation steps in the TwoL system.

Each composed module consists of a number of tasks where each task is either an activation of another composed module or of a BM. Composed modules do not directly contain operations on data, they only specify the structure of the task parallelism. Operations on data are hidden within the BMs. The tasks in a composed module may be executed in parallel (operator $||$) if there is no dependence between them or must be executed sequentially (operator $\circ$), otherwise. Parallel and sequential loops with an arbitrary nesting can also be used to construct composed modules. Thus, a TwoL program consists of a hierarchically structured level of task parallelism which is specified by the module specification and a flat level of data parallelism which is specified within the BMs.

This clear separation is used to divide the work between the programmer and the compiler system. The programmer specifies an algorithm as a module specification which expresses the maximum degree of task parallelism and provides a specification of the data parallel BMs. The compiler translates the module specification into a task parallel program that performs independent tasks within composed modules by independent groups of processors, if this leads to a smaller runtime than a pure data parallel execution. Data parallel executions are included as calls to BMs. Redistributions are inserted if necessary. Figure 1 gives an overview of the structure of the model. The specifications of the available degree of parallelism is shown in the upper part. (BM describes data parallelism within basic modules, M describes task parallelism by the module specification.) The translation into a full specification of a parallel program with mixed task and data parallelism (including the scheduling of independent tasks and the assignment of processor groups to tasks) is shown in the lower left part. The final C+MPI program generated is shown in the right part. The interactions between the task parallel and the data parallel parts are described in the next subsection. The generation of the final message-passing program will be considered in Section 3.

2.2 Program derivation

To demonstrate the specification and the derivation process, we consider an algorithm from numerical analysis, the conjugate gradient (CG) method, which is a Krylow-space method to solve a

$r_0 := b - Ax_0;$
$p_0 := r_0;$
$p_{-1} = 0;$
for $(k = 0;\ k < K;\ k{+}{+})$ {
  (1) $w_k = Ap_k;$
  (2) $\xi_k := \langle p_k, r_0 \rangle / \langle w_k, p_k \rangle;$
  (3) $x_{k+1} := x_k + \xi_k p_k;$
  (4) $r_{k+1} := r_k - \xi_k p_k;$
  (5) $\mu_k := \langle w_k, w_k \rangle / \langle w_k, p_k \rangle;$
  (6) $\nu_k := \langle w_k, w_{k-1} \rangle / \langle w_{k-1}, p_{k-1} \rangle;$
  (7) $p_{k+1} := w_k - \mu_k p_k - \nu_k p_{k-1};$
}

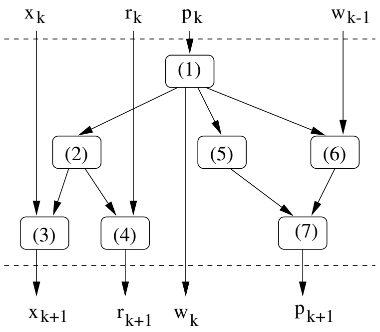

Figure 2: The conjugate gradient method.

system of linear equations $Ax = b$ with a symmetric, positive-definite coefficient matrix $A \in I\!R^{M \times M}$ [32]. In each step k, $0 \leq k \leq K-1$, the method chooses a search direction $p_k \in I\!R^n$ such that p_k is A-conjugate to $p_0, \ldots, p_{k-1}$, i.e., $p_l^T A p_k = 0$ for $l < k$. Figurs 2 shows a description of the algorithm (left) from [32] and the data flow (right) through one iteration step of the method. Figure 3 outlines the information flow for a detailed specification of one iteration step. This specification is a stepwise refinement of the initial one. Some technical details like storage of values used in the next iteration step (icons labeled $\mathtt{w}_{k-1}$, λ_{k-1}, $\mathtt{p}_{k-1}$) were also added. The detailed specification in Figure 3 suggests the potential of task parallelism of the algorithm for one iteration step. Figure 4 shows a specification of a composed module realizing the iteration steps for a fixed number of iterations $K < M$. The various parts of the module will be described in the following. (In practice, additional tests on the size of the residual r_k and a preconditioning will be used. Furthermore, due to rounding errors, a higher number of iteration, usually 2M-5M, is performed [28].)

A module specification is a non-executable program which only indicates the degree of parallelism without giving an exact execution order of tasks or specifying a data distribution for variables. A specific module contains activations of BMs where each BM (in Figure 1 on the top) describes the maximum degree of data parallelism of a specific computation without considering a specific target machine. The example in Figure 4 uses BMs `vv_add()`/`vv_sub()` for vector addition/subtraction, `sv_prod()` for the multiplication of a vector with a scalar, `vv_prod()` for the computation of a scalar product of two vectors, `mv_prod()` for the computation of a matrix-vector product, `op_assign()` for the assignment of the result of an arithmetical operation, and `vv_assign()`/`assign()` for the assignment of a vector/scalar to another vector/scalar. A specification of BM `mv_prod()` is given in Section 4.

The next step in the program derivation process for a given module specification is to make the design decisions (Figure 1, step M$\rightarrow$PM) and to transform the module specification into a non-executable *parallel module specification* (PM). An interactive compiler is responsible for selecting the most important design decisions for a parallel implementation: It decides whether the task parallelism of the module specification should be exploited (task scheduling) and how many of the available processors should be assigned to each BM. Moreover, it selects a data distribution for every variable of each BM such that the resulting communication is minimized. All decisions are based on the derivation of runtime formulas containing parameters that describe the relevant properties of the target machine, see [25, 10] for a detailed discussion. Fixing the implementation decisions leads to a parallel *frame program* exactly expressing the degree of parallelism that should be exploited for a given DMM. The frame program has the same structure as the module specification but contains information about data distribution, assignment of processors to modules, and additional operators to specify sequential execution of independent tasks in composed modules.

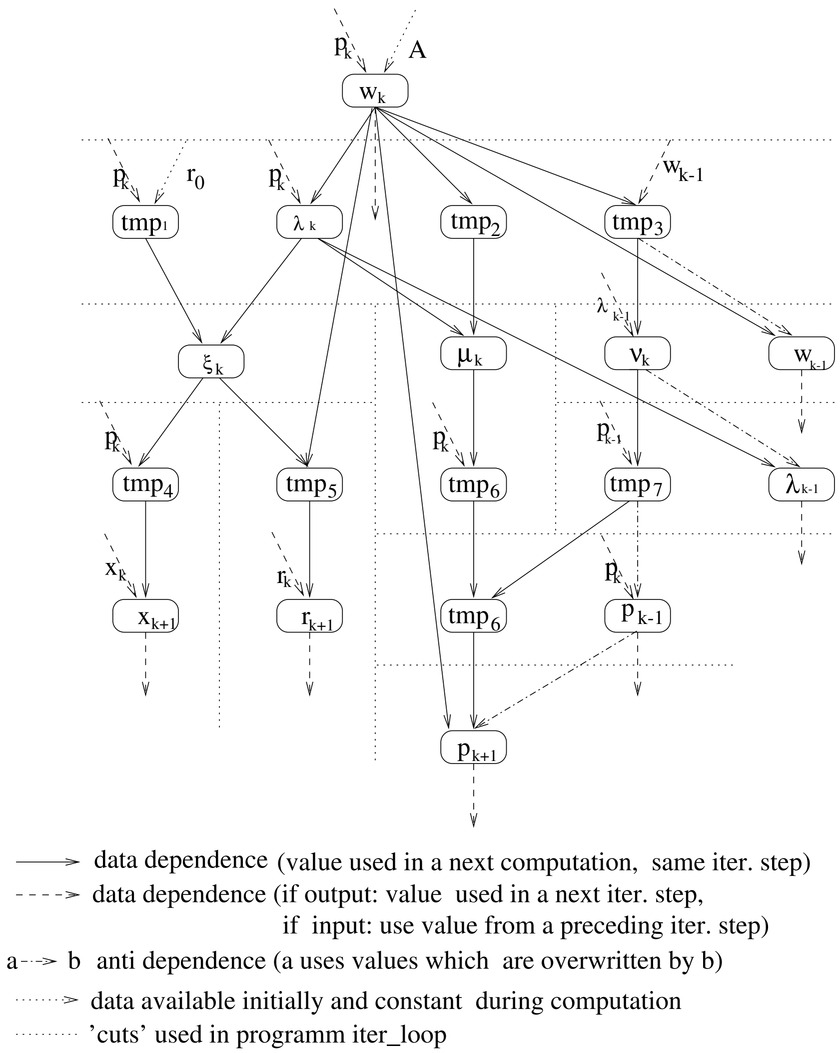

Figure 3: One iteration step of the conjugate gradient method.

Some additional requirements like a predefined input/output data distribution for the arguments of an inner module may be added. The system automatically determines the relevant data for the inner modules and inserts data redistribution operations if necessary.

For the translation of the composed module `iter_loop()` from Figure 4 to a frame program, a row-block distribution is chosen for matrix A and vector p is held replicated. In the frame program, all available processors will be used for the execution of the `w=mv_prod()` call. The resulting output vector w is block distributed and has to be redistributed (replicated) for the next computation steps. Then, four groups of processors of equal size are used to compute the scalar products $\mathtt{tmp_1} = \langle p_k, r_0 \rangle$, $\lambda_{\mathtt{k}} = \langle w_k, p_k \rangle$, $\mathtt{tmp_2} = \langle w_k, w_k \rangle$, and $\mathtt{tmp_3} = \langle w_k, w_{k-1} \rangle$, respectively. After these computations, two groups of processors are built to compute larger parts of the loop body, one group to compute the next approximation x_{k+1} and residual r_{k+1}, the other to compute the next search direction p_{k+1}. Since both computations roughly require the same number of computations, an equal splitting of the processor groups is appropriate. Although both computations have additional potential for a parallel execution, this will not be exploited for the frame program since the granularity of the computations is not very coarse, i.e., the overhead for establishing the concurrent execution would not be worth the effort on most DMMs for reasonable sizes of n.

```
module iter_loop (IN double A[M][M], double r_0[M], double p_0[M],
                  OUT double x[M])
(
   for (k=0 ..  K)
   (  mv_prod (A, p, w)
      ∘  ( vv_prod (p, r_0, tmp_1)
           || vv_prod (w, p, λ_1)
           || vv_prod (w, w, tmp_2)
           || vv_prod (w, w_0, tmp_3)
         )
      ∘  (  (   op_assign (tmp_1, div_op, λ_1, ξ)
                ∘  (  (   sv_prod (ξ, p, tmp_4)
                          ∘ vv_add (x, tmp_4, x)   )
                      ||  (   sv_prod (ξ, w, tmp_5)
                              ∘ vv_sub (r, tmp_5, r)   )
                   )
            ) || (  (  ( op_assign (tmp_2, div_op, λ_1, μ)   ∘ sv_prod (μ, p, tmp_6) )
                       ||  (   ( op_assign (tmp_3, div_op, λ_0, ν)  ||   vv_assign (w, w_0) )
                               ∘ ( sv_prod (ν, p_0, tmp_7) || assign (λ_1, λ_0) )
                           )
                    )
                    ∘  ( vv_add (tmp_6, tmp_7, tmp_6) || vv_assign (p, p_0) )
                    ∘  vv_sub (w, tmp_6, p)
                )
         )
   )
)
```

Figure 4: Composed module for the conjugate gradient iteration.

Figure 5 shows a parallel frame program for the conjugate gradient iteration for 4 processors (in order to make the example more readable we left out the data type specification for all variables). There is only one processor available for the computation steps labeled tmp_1, λ_k, tmp_2, and tmp_3. Thus, these BM calls will be executed sequentially. The operator ; specifies sequential execution of independent modules. The last parameter of each module specifies the processor group to be used for the specified module call. For modules where no explicit data distribution is specified, it is assumed that the variable is locally available on the processors specified by the PROC parameter. Modules performing data redistribution are also added to the parallel frame program.

After the design decisions are taken, the number of processors available for each BM activation and the input/output distribution for each variable of each BM are known. Thus, each call of a BM can be replaced by a call of a corresponding *parallel basic module* (PBM) representing the data parallel realization of the BM for a predefined number of processors and using a well-specified distribution of the variables passed as arguments. The PBMs are available in a library. A specific PBM library entry may provide additional information about runtimes which may be used for the design decisions on the upper level. The user also has the possibility to influence the system decisions, e.g., by designating a certain processor group to be used for a specified PBM call. Such requirements influence the antecedent step of the derivation of a frame program, e.g., since additional data redistribution may be necessary.

The program derivation process ends with an automatic implementation step, where parallel frame programs are translated, for example, into C program sequences with MPI communication operations (step PM→C+MPI in Figure 1). Figure 6 shows an extract of the C+MPI program generated by the TwoL system for the frame program from Figure 5. Section 3 describes the corresponding transformations in more detail. Calls to PBMs are replaced by calls to the corresponding library functions. These could be functions from a predefined library like ScaLAPACK or self-

```
module iter_loop ( IN A ROW_BLOCK(1··4), p REPLIC(1··4), x LOCAL(1),
                   r0 LOCAL(1), p0 LOCAL(4), λ0 LOCAL(4), w0 LOCAL(4),
                   OUT x LOCAL(1), PROC 1··4 )
for (k = 0··K) {
   mv_prod ( IN A ROW_BLOCK(1··4), p REPLIC(1··4),
             OUT w ROW_BLOCK(1··4), PROC 1··4 )
   ∘
   redistribute ( IN w LOCAL(1··4), OUT w REPLIC(1··4), PROC 1··4 )
   ∘
   {  vv_prod ( IN p, r0, OUT tmp1, PROC 1 )
      || vv_prod ( IN w, p, OUT λ1, PROC 2 )
      || vv_prod ( IN w, w, OUT tmp2, PROC 3 )
      || vv_prod ( IN w, w0, OUT tmp3, PROC 4 )
   }
   ∘
   {  redistribute ( IN λ1 LOCAL(2), OUT λ1 REPLIC(1··4), PROC 1··4 )
      ; redistribute ( IN tmp1 LOCAL(1), OUT tmp1 REPLIC(1··2), PROC 1··2 )
   }
   ∘
   {  {  op_assign ( IN tmp1 REPLIC(1··2), div_op, λ1 REPLIC(1··2),
                     OUT ξ REPLIC(1··2), PROC 1··2 )
         ∘
         {  {  sv_prod ( IN ξ, p, OUT tmp4, PROC 1 )
               ∘ vv_add ( IN x, tmp4, OUT x, PROC 1 ) }
            ||
            {  sv_prod ( IN ξ, w, OUT tmp5, PROC 2 )
               ∘ vv_sub ( IN r, tmp5, OUT r, PROC 2 ) }
         }
      }
      ||
      {  {  {  op_assign ( IN tmp2, div_op, λ1, OUT μ, PROC 3)
               ∘ sv_prod ( IN μ, p, OUT tmp6, PROC 3 ) }
            ||
            {  {  op_assign ( IN tmp3, div_op, λ0, OUT ν, PROC 4)
                  ; vv_assign ( IN w, OUT w0, PROC 4 ) }
               ∘
               {  sv_prod ( IN ν, p0, OUT tmp7, PROC 4)
                  ; assign ( IN λ1, OUT λ0, PROC 4 ) }
            }
         }
         ∘
         redistribute ( IN tmp7 LOCAL(4), OUT tmp7 REPLIC(3··4), PROC 3··4 )
         ∘
         {  vv_add ( IN tmp6, tmp7, OUT tmp6, PROC 3 )
            || vv_assign ( IN p, OUT p0, PROC 4 ) }
         ∘ vv_sub ( IN w, tmp6, OUT p, PROC 3 )
      }
   }
   ∘ redistribute ( IN p LOCAL(3), OUT p REPLIC(1··4), PROC 1··4 )
}
end
```

Figure 5: Parallel frame program (version with 4 processors) for the conjugate gradient iteration.

written functions. If executable code for PBMs are not available, the BMs have to be expressed in a suitable specification. Those specifications are first transformed into a PBM according to the requirements from the upper level (step BM→PBM, Figure 1). As an example, a specification of the PBM `mv_prod()` is given in Section 4. Thereafter, the specification is translated into the final C program with MPI communication operations (step PBM→C+MPI). An approach for the specification of the data parallel level is outlined in Section 4.

3 Task Parallel Execution

In this section we give a more detailed description of the task parallel coordination language and show how it can be translated into a task parallel program.

3.1 Description of task parallel executions

Module specification As illustrated by the example in Section 2, the potential degree of task parallelism of an algorithm can be described by a *module specification* which allows the specification of parallel or sequential execution of program parts. A module specification consists of composed modules and BMs. BMs denote program parts to be realized exploiting data parallelism as described in Section 4. Composed modules contain calls to other (composed or basic) modules and, hence, allow expressing arbitrary hierarchies of task parallelism. A module specification can be considered as a coordination program that expresses data dependencies between module activations and that separates task and data parallel parts.

The module specification is expressed in a coordination language which is based on *module expressions*. The syntax uses a set of variables for BMs $\mathcal{B}_1, \ldots, \mathcal{B}_l$, a set of variables for composed modules $\mathcal{M}_1, \ldots, \mathcal{M}_m$, and a set of variables $x, y, z, \ldots$ for scalars, vectors, matrices, or higher dimensional global data objects. Variables for basic modules and composed modules have a range and image of fixed arity and type. Module variables $\mathcal{M}$ are defined by module expressions

$$\mathcal{M}_i(x_{1_i}, \ldots, x_{m_i}) = T_i, \qquad i = 1, \ldots, n, \tag{1}$$

where each module expression T_i is composed of operators that coordinate the execution of BMs and other composed modules:

$$\begin{array}{rcl} T & ::= & \mathcal{B}_i(y_{1_i}, \ldots, y_{m_i}) \mid \mathcal{M}_i(x_{1_i}, \ldots, x_{m_i}) \mid T_1 \circ T_2 \mid T_1 \parallel T_2 \\ & \mid & \text{for } (i = 0, \ldots, n)\ \mathcal{M}(i, x_1, \ldots, x_m) \\ & \mid & \text{parfor } (i = 0, \ldots, n)\ \mathcal{M}(i, x_1, \ldots, x_m) \\ & \mid & \text{while } (i, \mathit{cond})\ \mathcal{M}(i, x_1, \ldots, x_m) \\ & \mid & \text{if } (\mathit{cond}) \text{ then } T_1 \text{ else } T_2 \end{array} \tag{2}$$

Data variables $(x_{1_i}, \ldots, x_{m_i})$ may be used as arguments to module expressions T_i. The meaning of the operators in T_i is as follows: $\circ$ denotes a sequential composition of modules, $\parallel$ denotes a potential for a concurrent execution of modules, *for* and *while* denote sequential loops, *parfor* denotes a parallel loop, *if* denotes a control dependence. The occurrence of a BM name in a module expression can be considered as a call of a corresponding function that is provided from outside.

Frame program A module specification is not executable since it only specifies the degree of parallelism but it gives neither an exact execution order of tasks nor a data distribution for variables. Now we consider a specific target machine with a given number of processors and known communication behavior. A module specification is transformed into a frame program by fixing the distribution of variables, the scheduling of modules, and the assignment of groups of processors to

$expression$	$\mathcal{T}(expression)$
$\mathcal{M}_1(x_{1_1},\dots,x_{m_1}) \;=\; T_1$ $\vdots$ $\mathcal{M}_l(x_{1_l},\dots,x_{m_l}) \;=\; T_l$	$\mathcal{T}(\mathcal{M}_1(x_{1_1},\dots,x_{m_1}) \;=\; T_1)$ $\vdots$ $\mathcal{T}(\mathcal{M}_l(x_{1_l},\dots,x_{m_l}) \;=\; T_l)$
$\mathcal{M}_k(x_{1_k},\dots,x_{m_k}) = T_k$	`void` $M_k(type(x_{1_k}),\dots,type(x_{m_k}),$`MPI_Comm comm`) $\{\mathcal{T}(T_k, MFG(M_l), comm)\}$
$\mathcal{B}_i(y_{1_i},\dots,y_{m_i})$	$\mathcal{V}(\mathcal{B}_i, MFG(M_l))(y_{1_i},\dots,y_{m_i},$ `comm`);
$\mathcal{M}_i(x_{1_i},\dots,x_{m_i})$	$\mathcal{M}_i(x_{1_i},\dots,x_{m_i},$ `comm`)

$T_1 \circ T_2$	$\mathcal{T}(T_1,$ `comm`);
	`MPI_Comm_dup (comm,&newcomm);`
	$\mathcal{T}(T_2,$ `newcomm`);
	`MPI_Comm_free(&newcomm);`

$T_1 \parallel T_2$	`MPI_Comm_rank(comm,&myrank);`
	`g`$_1$=$\mathcal{A}(MFG,T_1)$; `g`$_2$=$\mathcal{A}(MFG,T_2)$;
	`if (myrank <` `g`$_1$`) color=0; else color=1;`
	`MPI_Comm_split(comm,color,myrank,&newcomm);`
	`if (myrank <` `g`$_1$`)` $\mathcal{T}(T_1,$ `newcomm`); `else` $\mathcal{T}(T_2,$ `newcomm`);
	`MPI_Comm_free(&newcomm);`

for $(i = 0,\dots,n)$
$\mathcal{M}(i, x_1,\dots,x_m)$
	`for (i=0;i<n;i++) {`
	`MPI_Comm_dup(comm,&newcomm[i]);`
	$\mathcal{T}(\mathcal{M}(i, x_1,\dots,x_m)$, MFG, `newcomm[i]`);
	`MPI_Comm_free(&newcomm); }`

parfor $(i = 0,\dots,n)$
$\mathcal{M}(i, x_1,\dots,x_m)$
	`MPI_Comm_rank(comm,&rank);`
	`for (i=0,sum[0]=0;i<n;i++) {`
	`g[i]=` $\mathcal{A}$(MFG,$\mathcal{M}(i,\dots)$); `sum[i+1]=sum[i]+g[i]; }`
	`for (i=0;i<n;i++)`
	`if ((rank>=sum[i])&&(rank<sum[i+1])) color=i;`
	`MPI_Comm_split(comm,color,rank,&newcomm);`
	`MPI_Comm_rank(newcomm,&newrank);`
	`for (i=1;i<n;i++) {`
	`if ((rank >= sum[i]) && (rank < sum[i+1]))`
	$\mathcal{T}(\mathcal{M}(i, x_1,\dots,x_m)$, MFG, `newcomm`); }
	`MPI_Comm_free(&newcomm);`

Table 1: Translating module expressions of a frame program into task parallel programs in C using MPI.

module activations. A frame program has the same structure as a module specification (1), (2) but may contain the additional operators ';' (sequential execution of independent module activations) and *seqfor* (sequential loop over independent module activations). These operators are used in the frame program either if the target machine does not provide enough processors to execute all modules in parallel or if the communication behavior suggests to execute independent module activations consecutively (although they might be executed concurrently). The number of processors used for the execution of a BM is expressed in the frame program by specifying the number of processors for each module call. Additionally, for each variable on the left or right hand side of a module expression, a data distribution type is used that indicates the distribution that has been chosen for this variable.

After the scheduling and the assignment of processors to modules, the number of processors that are available for the execution of each BM activation is given. Thus, for each call of a BM in the frame program, a concrete PBM is chosen from a library. This PBM realizes the BM for the given number of processors and for the given data distribution of the variables that are passed as arguments. Using a C+MPI program corresponding to the PBM, a frame program can be translated into a task parallel C program. This translation will be considered in the next subsection.

3.2 Implementing parallel frame programs

Groups size information and data distribution information is stored in the global *module flow graph* (MFG) for a specific module specification M_l (M_l is the main symbol of the module specification). MFG(M_l) is a graph with one node for each occurrence of a module name in a module expression. The edges show relations between calling modules and called modules. The global MFG results by an expansion of the local graph for M_l by substituting the module variables by their corresponding definitions. Thus, the global MFG contains only calls to BMs. A hierarchical graph structure (where nodes for modules correspond to pointer to the expansion and annotations for group size and data distribution information) provides all information of the parallel design decisions. For the final parallelization the parallel design decisions are to be realized in an executable message passing program, e.g., in C using MPI. The translation function $\mathcal{T}$ is recursively defined and maps module syntax, MFG information, and communication group specification onto a message passing program, i.e., a syntax-directed translation is combined with additional parallelization information.

Table 1 summarizes the translation: $\mathcal{A}(\text{MFG}, \mathcal{M}_l)$ denotes the group size information. A sequence of module expressions $\mathcal{M}_1, \ldots, \mathcal{M}_l$ is translated into a sequence of corresponding function declarations with the same input/output variables as the composed module and an additional parameter for the communication context provided by the calling module. The communication context is realized by the communicator mechanism of MPI [27]. The communication context passed as parameter will be used for all message transmissions within the corresponding functions. For each BM, we assume that the data parallel part provides a PBM and a corresponding parallel C function. Since different data distributions may be realized for the input/output variables, different parallel versions of the same BM may be provided. The translation function $\mathcal{T}$ selects for each call of $\mathcal{B}_i$ the corresponding version $\mathcal{V}(\mathcal{B}_i, MFG(M_l))$ according to the data distribution of the input variables at the calling point. If no fitting PBM is provided, a redistribution has to be performed. Redistribution functions are treated in the frame program in the same way as PBMs.

The occurrence of a module variable in a module expression is translated into a call to the corresponding module function. The sequential composition of module expressions is translated into a sequential execution of the corresponding statements with a duplication of the communicator between these parts in order to separate the internal message transmissions. The parallel composition of module expressions is translated into a task parallel execution of the corresponding statements. This is obtained by splitting the communication context according to the information in the MFG and by executing the corresponding program parts by exactly those processors that belong to the processor group associated with the communicator. The correct context for data transmission is guaranteed by the use of the split communicator. The translation of a *parfor* construct is a generalization of the translation for the concurrent composition. Similarly, the translation of a sequential *for*/*seqfor* loop is a generalization of the translation for the sequential composition.

Figure 6 shows a part of the message-passing coordination program that is generated by the prototype of a compiler tool according to the translation scheme of Table 1. The fragment shown corresponds to lines (22-28) of the frame program of Figure 5.

```
        ⋮
/* lines (22-28) */
MPI_Comm_dup (newcomm1, &newcomm2);
{ /* lines (23-27) */
   MPI_Comm newcomm3;
   int g[2];
   int sum[3];
   int color;
   int i;

   MPI_Comm_rank (newcomm2, &myrank);
   g[0] = max (amfg ("sv_prod"), amfg ("vv_add"));
   g[1] = max (amfg ("sv_prod"), amfg ("vv_add"));
   for (i = 0, sum[0] = 0; i < 2; i++)
      sum[i + 1] = sum[i] + g[i];
   for (i = 0; i < 2; i++)
      if ((sum[i] <= myrank) && (myrank < sum[i + 1]))
         color = i;
   MPI_Comm_split (newcomm2, color, myrank, &newcomm3);
   if ((sum[0] <= myrank) && (myrank < sum[1]))
      { /* lines (23-24) */
         MPI_Comm newcomm4;
         MPI_Comm_dup (newcomm3, &newcomm4);
         sv_prod (xi, p, tmp4, newcomm4);
         MPI_Comm_free (&newcomm4);
         MPI_Comm_dup (newcomm3, &newcomm4);
         vv_add (x, tmp4, x, newcomm4);
         MPI_Comm_free (&newcomm4);
      }
   else
      { /* lines (26-27) */
         MPI_Comm newcomm4;
         MPI_Comm_dup (newcomm3, &newcomm4);
         sv_prod (xi, w, tmp5, newcomm4);
         MPI_Comm_free (&newcomm4);
         MPI_Comm_dup (newcomm3, &newcomm4);
         vv_sub (r, tmp5, r, newcomm4);
         MPI_Comm_free (&newcomm4);
      };
   MPI_Comm_free (&newcomm3);
};
MPI_Comm_free (&newcomm2);
        ⋮
```

Figure 6: Fragment of the coordination program to the frame program of Figure 5.

4 Data Parallel Execution

The BMs may be provided by the programmer as executable (data parallel) programs, i.e., as PBM, or as a high-level specification of data parallel executions. In the first case, the PBM provided are called from the task parallel level. In the second case, the specification must be transformed into a PBM, i.e., by explicitly exploiting data parallelism. A high-level specification of a BM describes the maximum degree of data parallelism of a specific computation without considering a specific target machine. Depending on the specific algorithm to be implemented, a BM can contain an arbitrary number of computations. A BM has an explicitly defined input/output behavior and exactly specifies the operations to be performed. These operations determine the semantics of the BM. In the following, we use a UNITY-like notation for the high-level specification of BMs [6, 32].

A BM specification contains a *header* specifying the input/output parameters, a *declaration* section introducing local variables, and an *assignment* section consisting of a sequence of assignment statements. These assignment statements may be executed concurrently (expressed by $\|$) to each other or sequentially (expressed by ;) one after another according to the following specification language for assignment statements:

$$
\begin{array}{lll}
\texttt{assign\_stm} & \texttt{::=} \ \texttt{statement} \ \{ \ \texttt{;} \ \texttt{statement} \ \} & \\
 & \ \ \mid \ \texttt{statement} \ \{ \ \| \ \texttt{statement} \ \} & \\
\texttt{statement} & \texttt{::=} \ \langle \ \| \ \texttt{quantif} \ \texttt{::} \ \ \texttt{assign} \ \rangle & (quantified\ assignment) \\
 & \ \ \mid \ \langle \ op \ \ \texttt{quantif} \ \texttt{::} \ \ \texttt{expr} \ \rangle & (quantified\ expression) \\
 & \ \ \mid \ \texttt{variable\_list} \ \texttt{:=} \ \texttt{expr\_list} & (enumerated\ assignment) \\
\texttt{quantif} & \texttt{::=} \ \texttt{variable\_list} \ \texttt{:} \ \ \texttt{boolean\_expr} &
\end{array}
$$

A *quantified assignment* is a special form of a parallel assignment expressed by $\langle \| \ i_1, \ldots, i_n : q :: a \rangle$ where quantification q consists of definitions for ranges of bound variables $i_1, \ldots, i_n$ and boolean expressions. The boolean expressions may depend on bound variables, values of program variables, and constants. The term a may be any assignment and is executed for each value of the bound variables $i_1, \ldots, i_n$ for which q is true.

A *quantified expression* is a special quantified assignment expressed by $\langle op \ i_1, \ldots, i_n \ : \ q \ :: \ e \rangle$, where *op* is a binary, associative, commutative operator like addition or intersection and e is an expression containing $i_1, \ldots, i_n$. The value of the expression defined in this way is the result of applying operator *op* to the set of expressions obtained by evaluating e for all values of $i_1, \ldots, i_n$ for which q is true. For example,

$\langle +i : 0 \leq i \leq N :: a[i] \rangle$ specifies the expression $\sum_{i=0}^{i=N} a[i]$.

An *enumerated assignment* assigns the values of simple or conditional expressions on the right of := to corresponding variables listed on the left of :=. Number and type of variables have to match the corresponding expressions. For example, a specification of a BM `mv_prod()` for a matrix-vector multiplication of a matrix A with a vector v and result vector w is given in the following way:

BM `mv_prod (IN double` $A[M][M]$`, double` $v[M]$`, OUT double` $w[M]$`)`
assign
$\langle \| \ i : \ 0 \leq i < M :: \ w[i] := \langle +j : \ 0 \leq j < M :: \ A[i,j] \cdot v[j] \rangle \ \rangle$.

In order to derive a program for a specific parallel machine with p processors or a given subset of p processors, a BM has to be transformed into a *parallel basic module* (PBM). A PBM results from a BM by a parallelization, i.e., the PBM contains an additional quantified assignment over all processors at the outermost assignment level expressing for each processor p the operations to be executed by p. The PBM header denotes the set of processors involved in the parallel computation. Moreover, a distribution for each (input/output and local) variable has to be chosen and the resulting communication operations have to be inserted. Data distributions are described by *data distribution types* (represented as symbolic names) where each type describes a well-defined data distribution function. Technically, the data distribution is accomplished by means of a bijective map of the global index i to a pair of indices (p, i_p), where p is the process identifier and i_p the local index. As an example we specify a PBM obtained from BM `mv_prod`:

PBM `mv_prod` `(IN double` $A[M][M]$ `ROW_BLOCK` $(1\cdot\cdot P)$,
`double` $v[M]$ `REPLIC` $(1\cdot\cdot P)$,
`OUT double` $w[M]$ `ROW_BLOCK` $(1\cdot\cdot P)$, `PROC` $1\cdot\cdot P$)
assign
$\langle\| \, p : \ 1 \leq p \leq P ::$
$\langle\| \, i_p : \ 0_p \leq i_p < M_p :: \ w[i_p] := \langle + j_p : \ 0 \leq j_p < M :: \ A[i_p, j_p] \cdot v[j_p]\rangle \ \rangle \ \rangle.$

Since a PBM exactly specifies an SPMD computation, it can be transformed into a corresponding C program by converting the UNITY-like expressions to loops and by realizing the communication operations by a specific communication library like MPI. This transformation is similar to loop parallelization techniques [34, 13] and will not be considered further in this article.

5 Examples and Experiments

As examples, we consider the conjugate gradient method and solution methods for ordinary differential equations (ODEs). All methods considered have a potential of task and data parallelism, but it is not a priori clear, how the available parallelism should be exploited for an implementation of a specific parallel machine. We demonstrate, how the proposed model and translation scheme can be used for selecting the most efficient combination of task and data parallelism.

5.1 Conjugate gradient method

The CG method from Section 2 has been implemented in several variants using the prototype of the TwoL system: a task parallel variant, a data parallel variant, and a variant which mixes data and task parallelism. Figure 7 shows the speedups of the variants using 4 processors of a Convex SPP2000 and of a Cray T3E, respectively. As input the BCSSTM-matrices (no. 06, 07, 09, 11, and 13) of the Harwell-Boeing Collection have been used. The sparsity of the matrices has not been exploited for the implementation.

The data parallel execution of the CG method only exploits the existing potential of data parallelism, i.e., the module activations are executed consecutively by all processors available. The data parallel execution variant profits from the fact that for the CG method large data can mostly be keeped locally and only scalars have to be broadcasted through the system.

The task parallel version of the CG method is the implementation of the parallel frame program illustrated in Figure 5. The `mv_prod` procedure call is performed by all processors in a data parallel way. For 4 processors, implementing the program in a purely task parallel variant means that almost each of the remaining procedure calls, except the data redistribution operations, is executed by a single processor. The processors are synchronized each time when a data redistribution is performed.

The third version mixes data and task parallelism: the `mv_prod` and the first 4 `vv_prod` procedure calls are performed by all available processors in a data parallel way, the remaining of the program is purely task parallel.

The comparison in Figure 7 shows that the data parallel implementation leads to the largest speedup values on both the SPP2000 and the T3E. For the T3E, the difference between the variants is decreasing with the size of the matrix. The superlinear speedup values for the SPP2000 are due to caching effects.

5.2 Iterated Runge-Kutta methods

Iterated Runge-Kutta (RK) methods are one-step solution methods for initial value problems of ODEs of the form $\frac{dy(x)}{dx} = f(x, y(x))$, $y(x_0) = y_0$, $x_0 \leq x \leq x_{end}$ where $y : \mathbb{R} \rightarrow \mathbb{R}^n$ is the unknown solution function and $f : \mathbb{R} \times \mathbb{R}^n \rightarrow \mathbb{R}^n$ is an application-specific function which is nonlinear in the general case. The predefined vector $y_0 \in \mathbb{R}^n$ specifies the initial condition at x_0. The solution method computes approximation vectors $y_\kappa = \tilde{y}(x_\kappa)$ for the exact values $y(x_\kappa)$ at discrete x-values

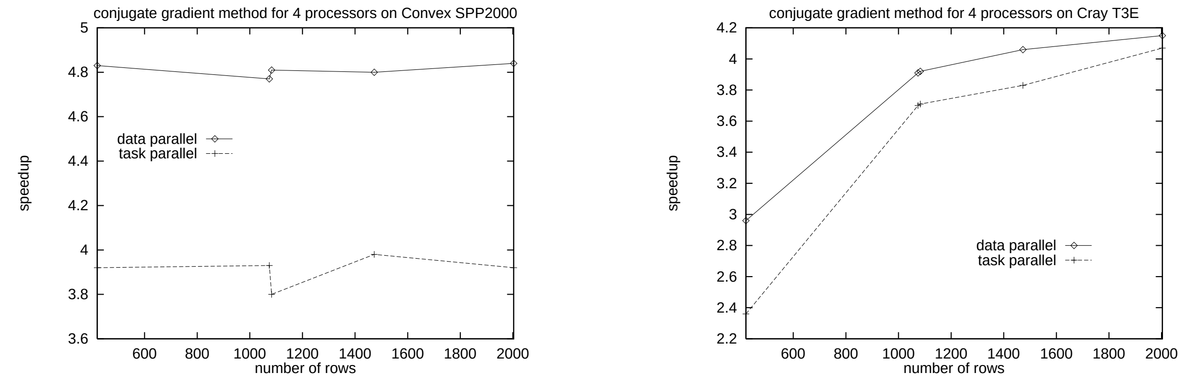

Figure 7: Speedups of task parallel and data parallel executions of the conjugate gradient method on a Convex SPP2000 (left) and Cray T3E (right).

$x_\kappa = x_{\kappa-1} + h$ one after another with *step-size* h. There is a large variety of RK methods that can be used for ODEs with different characteristics. Here, we consider RK methods with a large potential of task and data parallelism. These methods have been especially designed for a parallel execution [33, 22].

We apply the solution methods to two classes of ODEs which differ in the amount of computational work of the right hand side f of the ODE system:

- f has fixed evaluation costs that are independent of the system size (sparse function);
- the evaluation costs of f depend linearly on the system size (dense function).

Both cases may occur when solving systems of differential equations. The discretization of the spatial derivatives of a time-dependent partial differential equation results in a function f with a constant computational effort [15]. A function f with evaluation costs that depend on the system size arises, e.g., when solving nonlinear partial differential equations with Fourier–Galerkin methods [8]. In the experiments we used the Brusselator equation as example for a partial differential equation that results in a sparse system of ODEs. The Brusselator equation is a reactor-diffusion system from chemical kinetics that models the reaction of two chemical substances [15]. Depending on the parameter values in the Brusselator equation, the resulting ODE system is stiff or non-stiff, respectively.

In each time step, an s-stage iterated RK method performs a fixed number m of iterations to compute s stage vectors $v^1, \ldots, v^s$ iteratively and uses the result of the last iteration to compute the next approximation vector $y_{\kappa+1}^{(m)}$:

$$
\begin{aligned}
v_{(0)}^l &= f(y_\kappa), \qquad l = 1, \ldots, s, \\
v_{(j)}^l &= f(y_\kappa + h_\kappa \sum_{i=1}^{s} a_{li} v_{(j-1)}^i), \quad l = 1, \ldots, s, \quad j = 1, \ldots, m, && (3) \\
y_{\kappa+1}^{(m)} &= y_\kappa + h_\kappa \sum_{l=1}^{s} b_l \, v_{(m)}^l. && (4)
\end{aligned}
$$

The advantage for a parallel execution is that the iteration system (3) of size $s \cdot n$ consists of s independent function evaluations that can be performed concurrently in a task parallel way. Since each function evaluation requires the same execution time, a task parallel execution uses processor groups of the same size. Another possibility is a data parallel execution which performs the s function evaluations consecutively by all available processors. Figures 8 and 9 show the

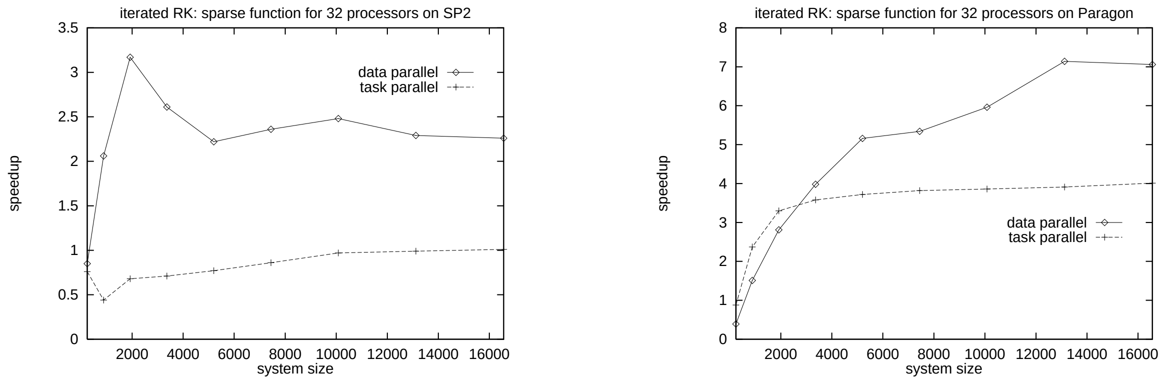

Figure 8: Speedups of task parallel and data parallel execution schemes of the iterated RK method on the IBM SP2 (left) and Intel Paragon (right), sparse input system.

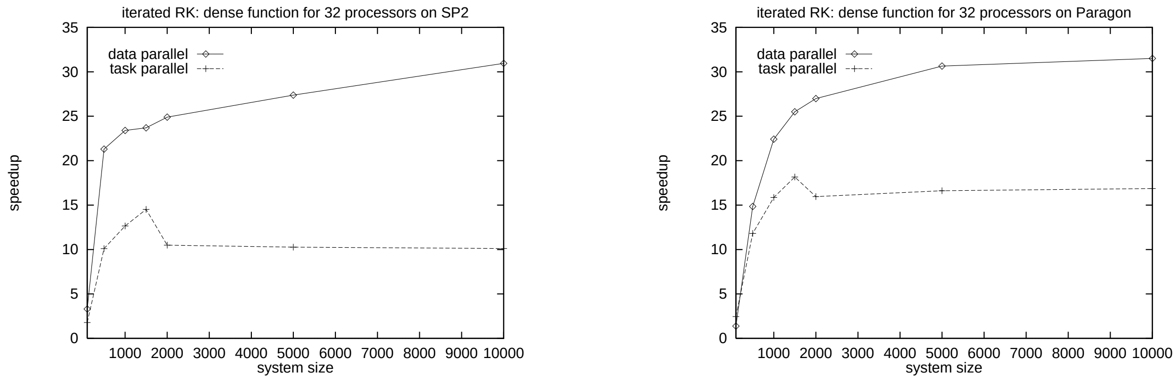

Figure 9: Speedups of task parallel and data parallel execution schemes of the iterated RK method on the IBM SP2 (left) and Intel Paragon (right), dense input system.

resulting speedup values of a three-stage iterated RK method when applied to a sparse or dense ODE system, respectively. The tests have been performed on an IBM SP2 and an Intel Paragon with 32 processors. The figures show that in most cases, the pure data parallel variant performs better than a variant that exploits the available task parallelism in a mixed task parallel and data parallel execution. There are two main reasons for this behavior. First, the change from the group computation of the iteration vectors to the global computation of the next approximation vector requires additional (global) communication the costs of which outweigh the savings by the group communication. Second, for 32 processors, the sizes of the resulting groups differ (two groups get 11 processors and one group gets 10 processors), thus causing a load imbalance.

5.3 Diagonally iterated implicit Runge-Kutta methods

Iterated RK methods are explicit solution method which are only suitable to solve *non-stiff* ODE systems. To provide RK-solver for stiff ODE systems exhibiting task parallelism, *diagonally iterated implicit RK* (DIIRK) methods have been introduced. The computation scheme of an s-stage DIIRK

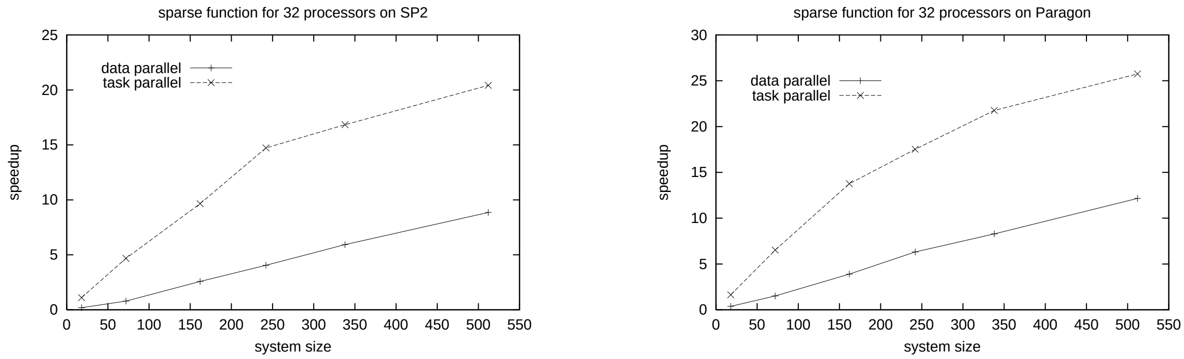

Figure 10: Speedups of task parallel and data parallel execution schemes of the DIIRK method on the IBM SP2 (left) and Intel Paragon (right), sparse input system.

method is similar to the computation scheme of an s-stage iterated RK method:

$$v_{(0)}^{l} = y_\kappa, \qquad l = 1, \ldots, s, \tag{5}$$

$$v_{(j)}^{l} = y_\kappa + h_\kappa \sum_{i=1}^{s} (a_{li} - d_{li}) f(v_{(j-1)}^{i}) + h_\kappa d_{ll} f(v_{(j)}^{l}), \quad l = 1, \ldots s, \quad j = 1, \ldots, m, \tag{6}$$

$$y_{\kappa+1}^{(m)} = y_\kappa + h_\kappa \sum_{l=1}^{s} b_l f(v_{(m)}^{l}). \tag{7}$$

For a system of ODEs of size n, Equation (6) denotes s implicit non-linear equation systems which are independent from each other since the computation of $v_{(j)}^{l}$ is independent from $v_{(j)}^{k}$ for $k \neq l$. For each non-linear equation systems, we use the Newton method to find a solution vector. The Newton method performs a number of iteration steps where each step includes the solution of a linear equation system. In our implementation, these systems are solved with a Gaussian elimination method.

In a task parallel execution of the DIIRK method, the s iteration vectors of one DIIRK iteration are computed by s independent groups of processors where each group contains about the same number of processors. It is convenient to implement the Newton method in such a way that the result is delivered replicated among the executing processors. Internally, a row cyclic distribution of the iteration vectors is used because this leads to a good performance of the Gaussian elimination method that is used within each iteration step of the Newton method. At the end of each DIIRK iteration, a redistribution has to be executed to make the iteration vectors available to all processors.

Figures 10 and 11 show the resulting speedup values of a five-stage DIIRK method when applied to a sparse or dense ODE system, respectively. The figures show that the task-parallel execution leads to a significantly better performance than a pure data parallel consecutive execution that solves the independent systems of one corrector step by all available processors one after another. The reason for this lies in the fact that the broadcast operations executed by the Gaussian elimination are less expensive when executed in parallel on independent groups of processors (group broadcast).

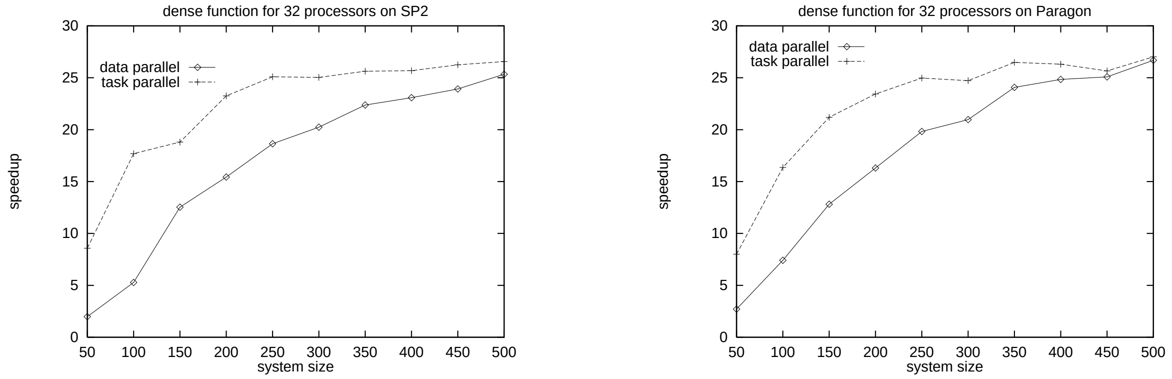

Figure 11: Speedups of task parallel and data parallel execution schemes of the DIIRK method on the IBM SP2 (left) and Intel Paragon (right), dense input system.

6 Comparison with Related Work

Work related to the TwoL approach includes studies on new parallel programming paradigms [18], computation models [1, 7, 31], performance prediction techniques [9, 17, 19], and parallelizing compilers. In the following, we compare our approach to similar approaches in the area of parallelizing compilers.

Several research groups working on parallelizing compilers have included support to combine task and data parallelism. Fortran M [11, 12] allows the creation of processes which can communicate with each other by predefined channels and which can be combined with HPF for a mixed task and data parallel execution. In contrast to the Fortran M approach, our approach is more directed towards the combination of fine and medium grain parallelism which is shown by the fact that different modules cannot communicate with each other. Moreover, the Fortran M compiler does not support the derivation of parallel implementation, i.e., the programmer must decide on the execution order of processes and on the data distributions by himself.

The Fx approach allows task parallelism by providing directives to partition processors into subgroups and to assign computations to different subgroups (task regions) [30]. Computations of a specific subgroup are executed in a data parallel way. The Fx compiler provides a mapping tool for the grouping of subroutine calls to modules and the mapping of processors to modules [29]. Although the Fx approach is similar in spirit to the TwoL approach, there are some important differences: The task regions in Fx cannot be nested lexically whereas TwoL allows a hierarchical structure of the 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 programming approach in which the programmer has to decide on the task partitioning and the assignment of task 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 programmer is responsible for specifying the available task parallelism, but the final decision whether the available task parallelism will be exploited and how the 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 [29] is based on *static* runtime expressions that do not take the structure of the subroutines into consideration and the parameter of which are determined by separate runtime tests for each application. A model that is similar to the task parallelism model of Fx has recently been added to High Performance Fortran [16] as an approved extension.

An exploitation of task and data parallelism in the context of a parallelizing compiler can be found in the Paradigm compiler [3, 19, 21]. The Paradigm compiler 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 [4]. Nodes in the macro data-flow graph correspond to basic parallel tasks or loop constructs, edges correspond to precedence constraints that exist between tasks. The nodes and edges are weighted with processing and data transfer costs both of which depend on the number of processors used for the execution. [19] describes scheduling and allocation algorithms for macro data-flow graphs where 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. [20] considers the generation of array redistributions between tasks.

There are two main differences between the 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 a different derivation procedure in both approaches. Second, the runtime prediction of Paradigm is based on measured execution times of the tasks which is similar to the Fx approach. This does not allow the use of the runtime prediction for the design of the task programs as it is possible when using the performance prediction method of TwoL.

7 Conclusions and Future Research

The simultaneous exploitation of task and data parallelism can lead to significantly faster programs than the sole exploitation of data parallelism [21, 24]. In this paper, we have outlined a general framework that allows the expression of task parallelism with a small coordination language as a module specification and that provides a clear separation and a clear interface between the task and the data parallel parts. Based on this separation, the module specification can be translated into a frame program which is a description of a parallel implementation of the task and data parallel parts. This translation can be performed in a user-guided way or automatically with a compiler tool that uses runtime formulas parameterized with a concise model of the target machine. The frame program can then be translated into message passing programs that are based on hierarchically structured communication contexts.

The advantage of this approach is not only that it provides a framework for a compilation system but also that it provides a systematic way of writing parallel programs with mixed task and data parallelism: The design decisions for exploiting parallelism are entirely separated from the generation of parallel code. Thus, the decisions can be realized with an interactive transformation tool on the internal representation of the module specification. The result of the transformation, the intermediate frame program, can be translated by a syntax directed pass in any imperative language augmented with a message passing library supporting groups. (For message passing libraries not supporting groups, collective communications on a subset of the processors have to be simulated in a more tedious way.) We have chosen C as the imperative programming language and the message passing interface MPI as an example.

Currently, a prototype for translating frame programs into message-passing programs is available. The BMs are assumed to be available as executable parallel functions. Future research includes the implementation of the generation of frame programs from module specifications according to the derivation steps in Subsection 2.2 and an extension of the model to capture pipelining between dependent modules as an additional source of parallelism.

References

[1] A. Alexandrov, M. Ionescu, K.E. Schauser, and 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.

[2] H. Bal and M. Haines. Approaches for Integrating Task and Data Parallelism. *to appear: IEEE Concurrency*, 1998.

[3] P. Banerjee, J. Chandy, M. Gupta, E. Hodge, J. Holmes, A. Lain, D. Palermo, S. Ramaswamy, and E. Su. The Paradigm Compiler for Distributed-Memory Multicomputers. *IEEE Computer*, 1995.

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

[5] L.S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R.C. Whaley. *ScaLAPACK Users' Guide.* SIAM, 1997.

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

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

[8] R. Dautray and J. Lions. *Mathematical Analysis and Numerical Methods for Science and Technology*, volume 2. Springer, 1990.

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

[10] R. Foschia, T. Rauber, and G. Rünger. Modeling the Communication Behavior of the Intel Paragon. In *Proc. 5th Symp. on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS'97)*, IEEE, pages 117–124, 1997.

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

[12] I. Foster, M. Xu, B. Avalani, and A. Choudhary. A Compilation System That Integrates High Performance Fortran and Fortran M. In *Proccedings 1994 Scalable High Performance Computing Conference.* IEEE Computer Society Press, 1994.

[13] H. Gerndt. *Automatic Parallelization for Distributed-Memory Multiprocessing Systems.* PhD thesis, Universität Bonn, 1989.

[14] M. Girkar and C. Polychronopoulos. Automatic Extraction of Functional Parallelism from Ordinary Programs. *IEEE Transactions on Parallel and Distributed Systems*, 3(2):166–178, 1992.

[15] E. Hairer and G. Wanner. *Solving Ordinary Differential Equations II.* Springer, 1991.

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

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

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

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

[20] S. Ramaswamy and P. Banerjee. Automatic Generation of Efficient Array Redistribution Routines for Distributed Memory Multicomputers. *Proc. Frontiers '95*, pages 342–349, 1995.

[21] S. Ramaswamy, S. Sapatnekar, and P. Banerjee. A Framework for Exploiting Task and Data Parallelism on Distributed-Memory Multicomputers. *IEEE Transactions on Parallel and Distributed Systems*, 1997.

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

[23] T. Rauber and G. Rünger. Load Balancing Schemes for Extrapolation Methods. *Concurrency: Practice and Experience*, 9(3):181-202, 1997.

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

[25] T. Rauber and G. Rünger. Deriving Structured Parallel Implementations for Numerical Methods. *Microprocessing and Microprogramming*, 41:589–608, 1996.

[26] T. Rauber and G. Rünger. The Compiler TwoL for the Design of Parallel Implementations. In *Proc. 4th Int. Conf. on Parallel Architectures and Compilation Techniques*, IEEE, pages 282–301, 1996.

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

[28] J. Stoer and R. Bulirsch. *Introduction to Numerical Analysis.* Springer, New York, 1990.

[29] 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.

[30] J. Subhlok and B. Yang. A New Model for Integrating Nested Task and Data Parallel Programming. *8th ACM SIGPLAN Symp. on Principles & Practice of Parallel Programming*, 1997.

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

[32] E.F. Van de Velde. *Concurrent Scientific Computing.* Springer, 1994.

[33] P.J. van der Houwen and B.P. Sommeijer. Parallel Iteration of high–order Runge–Kutta Methods with stepsize control. *Journal of Computational and Applied Mathematics*, 29:111–127, 1990.

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