A Coordination Language for Mixed Task and Data Parallel Programs

Thomas Rauber
Institut für Informatik
Universität Halle-Wittenberg
Kurt-Mothes-Str.1
06120 Halle (Saale), Germany
rauber@informatik.uni–halle.de

Gudula Rünger
Institut für Informatik
Universität Leipzig
Augustusplatz 10/11
04109 Leipzig, Germany
ruenger@informatik.uni–leipzig.de

Keywords: mixed task and data parallelism, coordination language, message-passing programs, parallel scientific computing.

ABSTRACT

We present a coordination model to derive efficient implementations using mixed task and data parallelism. The model provides a specification language in which the programmer defines the available degree of parallelism and a coordination language in which the programmer determines how the potential parallelism is exploited for a specific implementation. Specification programs depend only on the algorithm whereas coordination programs may be different for different target machines in order to obtain the best performance. The transformation of a specification program into a coordination program is performed in well-defined steps where each step selects a specific implementation detail. Therefore, the transformation can be automated, thus guaranteeing a correct target program. We demonstrate the usefulness of the model by applying it to solution methods for differential equations.

1. INTRODUCTION

Many application algorithms exhibit different kinds of potential parallelism. *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 [19], i.e., different iterations of a loop (performing the same operations) are executed by different processors on different data. *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 coarse granularity. However, 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 [10]. This is especially true for parallel machines with a large number of processors like the ASCI teraflop machines. Applications that can benefit from a combination of task and data parallelism include examples from numerical analysis, signal processing [17], and multidisciplinary codes like global climate modeling, see [1] for a good overview. Since many applications benefit from an exploitation of both task and data parallelism, many languages have been proposed to combine data parallel with task parallel executions, including Fx, Fortran M, Braid, Opus, or Orca [1]. A more detailed discussion is given later.

Usually, there are many possibilities for combining data parallel computations in a task parallel way, i.e., there is a large variety of alternative parallel implementations for one algorithm and it is often difficult to choose the most efficient variant. To obtain an efficient parallel implementation for a specific algorithm, the programmer has to take properties of the algorithm and of the target machine into account.

In this paper, we present a model for a systematic derivation of efficient message-passing programs with mixed task and data parallelism which separates the algorithmic properties from the properties of the target machine and partitions the derivation process of a parallel implementation for a given algorithm into several steps. Data parallel executions are expressed as modules that can be executed by a varying number of processors. Depending on their data dependencies, different modules can be executed concurrently to each other on disjoint groups of processors or must be executed consecutively. A *specification language* provides constructs to express possible execution orders between modules and thus describes the available degree of task parallelism explicitly. The transformation of the specification program into a *coordination program* describes how the available degree of parallelism is actually exploited for a specific parallel implementation. The result is a complete description of a parallel program that can easily be translated into a message-passing program. The coordination program is expressed in a coordination language which is an extension of the specification language. The final message-passing program is expressed in C with MPI (message-passing interface [6]). A coordination program can express only those parallel programs that are allowed by the corresponding specification program. So for example, if the specification program requires two modules to be executed consecutively, the coordination program is not allowed to execute them concurrently. Thus, based on a correct specification program, only correct coordination programs can result.

The main advantage of this approach is that it provides a basis for a systematic derivation of efficient parallel implementations. The derivation of the coordination program from a given specification program is further subdivided into several decision steps which serve as a basis for an automatization of the design process in an interactive compiler tool. The paper introduces the specification language to express mixed task and data parallelism and the coordination language to express the parallel design decisions.

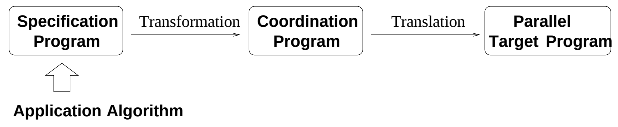


Figure 1: Macro-steps of the derivation process.

We demonstrate the expressiveness of the approach with examples from scientific computing. The languages are also the interface to a compiler tool and serve as the basis for a cost model.

The remainder of the article is organized as follows. Section 2 gives a brief overview of the general approach for the derivation of parallel programs. Section 3 describes the language support. Section 4 applies the approach to solution methods for differential equations. Section 5 discusses related work and Section 6 concludes.

2. GENERAL APPROACH

The derivation process for a parallel implementation of an algorithm is divided into three macro-steps, see Figure 1.

- In the first step, the application algorithm is formulated as a *specification program* expressing the maximum degree of task and data parallelism available. This step is completely independent from the target architecture and depends only on the control and data dependencies of the algorithm.
- In the second step, the specification program is transformed into a *coordination program* that expresses which degree of the parallelism will be exploited for the parallel implementation and which distribution will be used for the variables. A program-oriented cost model will be used to guide the transformations [12]
- In the third step, the coordination program is translated into an executable message-passing program realizing the decisions of the second step.

In the following, we give a brief and informal overview of these macro-steps. In Subsection 2.2 we sketch the cost model and in Subsection 2.3 we describe the design steps to construct a coordination program from a specification program. We also mention methods and algorithms based on cost formulas which we use to yield a specific design decision. The emphasis of this paper is to introduce the language support for the specification and coordination of parallel algorithms and to demonstrate its suitability for applications from scientific computing.

2.1 Task and data parallelism in the derivation process

Since task and data parallelism are orthogonal to each other and require different treatment for its exploitation, we have partitioned the derivation process of a parallel program into a task parallel and a data parallel level. For both levels, the derivation process proceeds according to the macro-steps described in Figure 1. However, the internal structure of the macro-steps for the two levels is different according to the specific needs.

Within each macro-step, both levels are clearly separated and connected only by specific information transfer, see Figure 2. The specification of data parallelism consists of definitions of *basic modules* (BMs) expressing operations on data that can be executed in a data parallel way. To provide a clear interface to the task parallel level, each definition of a BM includes a specification of its input/output behavior. On the task parallel level, the specification of parallelism is expressed as a *specification program* which expresses data dependencies between BM activations. The specification program can be structured hierarchically by the definition of composed modules (CMs) which can again call other CMs or BMs. The specification program essentially consists of calls to (composed or basic) modules and expresses whether the calls can be executed in parallel or whether a sequential execution is necessary because of data dependencies. For the flexibility of the transformation into a parallel implementation, it is important that both levels express the maximum degree of task and data parallelism, respectively, since only the parallelism expressed in the specification can be exploited in later phases.

The specification program is transformed into a coordination program which describes the task parallel characteristics of the parallel implementation. In particular, it fixes the execution order for module calls and specifies for each call of a module how many processors are used for its execution. Moreover, the data distribution of the composed variables used by the computations within the BMs is determined. Although the coordination program is not directly executable, it specifies the most important parallel design decisions and can therefore easily be translated into a corresponding message-passing program.

Based on the information in the coordination program about the number of executing processors and about the data distributions, the specification of a BM can be transformed into a parallel basic module (PBM). This PBM fixes the internal data distributions and contains the resulting communication operations. If a coordination program calls the same BM with different data distributions, different parallel versions of the PBM have to be provided. The correct version for a call is selected according to the data distribution needed. A PBM contains the number of executing processors as parameter and, thus, the same version can be called with different numbers of executing processors.

In the final step, a *parallel implementation* consisting of two parts is generated: A coordination program in C creates separate MPI communication contexts for module calls that are performed concurrently and contains calls of PBM as specified in the coordination program [4]. For each CM, a corresponding C-function is provided which is constructed according to the calling structure of the CM. If redistributions between consecutive module calls are necessary, because different module calls require the same variable with different distributions, appropriate calls to a redistribution library are inserted. The coordination program does not contain operations on data. The PBMs in the coordination program are realized as C programs with MPI communication operations as well. In addition to the interface parameters of the BMs, each PBM has a parameter specifying the number of executing processors and a parameter specifying the communication context for the internal communication.

2.2 Cost model

To derive a coordination program that corresponds to an efficient parallel target program for a given parallel machine, a cost model has to be used. Our general approach for the cost model are runtime functions that express computation and communication times of a parallel programm depending on several parameters. Runtime functions of basic modules are derived according to the internal computation and communication operations of the basic modules. Runtime formulas for composed modules are built up according to the hierarchical structure of the modules. Additionally, the costs of redistributions of variables between calls of basic modules is taken into account. Based on the potential parallelism expressed in the specification program the runtime costs of possible coordination programs can be expressed. The costs can be used to compare

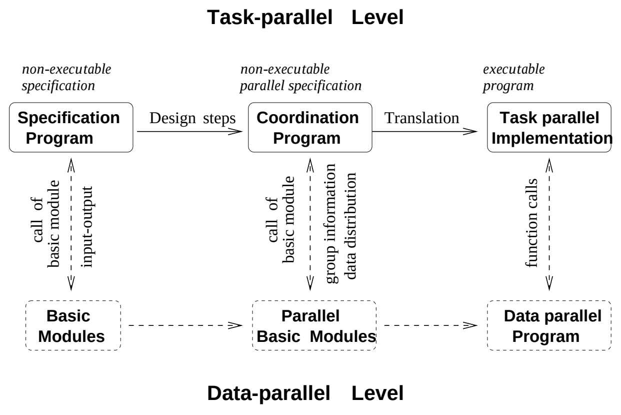


Figure 2: Design steps showing the task and the data parallel level

different parallel coordination programs for the same initial specification program and to select the most efficient one. But the costs can also be used as information to guide the design steps from the specification program to a coordination program.

The runtime functions are based on an abstract model of a parallel machine that uses a set of parameters to describe the behavior of the machine. The parameters include the number p of processors, the time t_{op} for the execution of an arithmetic operation *op*, the startup time τ for a message transfer, and the bandwidth b of the interconnection network. These parameters are used in the runtime functions to describe the execution time of basic operations and of collective communication operations in basic modules. The computation time is determined according to the computations specified in the basic module and uses application-specific parameters like iteration counts or sizes of data structures. The communication time depends on the internal data distribution of the basic module and describes the resulting internal communication operations. Each communication operation is described by a specific formula. For different data distributions, different communication operations with messages of different size might have to be performed and, hence, different runtime formulas may result for the same basic module.

2.3 Design decisions for mixed task and data parallelism

The derivation of a coordination program from a specification program is further subdivided into four substeps.

2.3.0.1 Execution order:.

In the first substep, the *execution order* of independent module calls is determined. These calls can either be executed concurrently by independent groups of processors or consecutively by all processors available. The decision on the execution order has to take the internal computation and communication structure of the BMs and the communication behavior of the target machine into consideration. For many parallel machines (including the IBM SP2, the Intel Paragon, or the Cray T3D and T3E), performing a collective communication operation on small groups of processors in parallel is less expensive than performing the same operation on all processors. This is due to the fact that, depending on the communication operation, there is a logarithmic or linear dependence of the execution time on the number of participating processors. Moreover, the effect of this property is increasing with an increasing number of processors. So, a concurrent execution of independent modules should be prefered. However, performing the computations by disjoint groups of processors might cause an additional overhead because of two reasons. First, we consider the situation that the module calls to be performed in parallel require an equal amount of computation. Nevertheless, the sizes of the processor groups working in parallel may differ since the number of processors available is not necessarily a multiple of the number of processor groups to be built. Hence, the computational work is not load balanced among the groups which possibly results in idle times. Second, building the processor groups takes a certain amount of execution time, as the corresponding communication contexts have to be established.

The decision how to realize the execution order can be described as a scheduling problem for multiprocessor tasks. Each activation of a basic module can be considered as a multiprocessor task which can be executed by a varying number of processors. The specification of parallelism determines possible dependencies between activations of multiprocessor tasks which makes an execution in a predefined order necessary. These data dependencies can be represented in a directed acyclic graph, the module dependence graph. The nodes of the graph are attached with the costs (in form of runtime formulas) of the corresponding module calls. Edges are annotated with redistribution costs if the internal data distributions of successive module calls require such a redistribution. A heuristic scheduling approach considering all possible group partionings for independent modules calls is applied to the multiprocessor task graph, see [14] for a detailed description.

2.3.0.2 Pipelining:.

In the second substep, data dependent modules are considered. For those modules there is the possibility to perform them on one group of processors one after another or to introduce a limited form of concurrent processing in the form of a *pipelined* execution. For a consecutive execution the next module is not started before the execution of the preceding module has been completed by all processors of the group. Depending on the data dependencies of the modules, their execution can be pipelined, i.e., the execution of the module calls happens in parallel and one module continuously produces data that is processed by the following module call. This is possible, if the first module produces its output data, i.e., data items of a compound data object, one after another in the same order in which the succeeding module call uses the data as input. Thus, the degree of parallelism can be increased since the modules can be executed in parallel by different processor groups. This is useful if the number of processors is large enough to execute both modules concurrently and if the use of all available processors for a consecutive execution of the modules would not increase the resulting performance because a saturation point of the speedup has been reached.

2.3.0.3 Processor groups:.

The result of the first two substeps is an exact execution order of the module calls, but the number of executing processors has not yet been determined. This is done in the next substep by considering the computational work of the different module calls and the resulting communication time. For a specific group g, the processors in g perform module calls concurrently to other groups such that at each point in time, all processors of g perform the same data parallel operations. Each group g has a certain lifetime starting with the generation of g and ending by merging g with other groups or further subdividing g. During its lifetime, each group has to perform module calls with internal computations. Considering groups working concurrently to each other, the best performance results, if the processors are partitioned among these groups according to the internal computation and communication time of the modules, i.e., if each group has the same parallel runtime. Thus, idle times are reduced as far as possible.

To avoid a load imbalance for modules to be executed concurrently the group sizes of the subsets of processors have to be adapted to the execution times of the modules. The problem can be considered as a constraint optimization problem as described in [13] for the extrapolation method. Usually, an approximation method has to be employed for the solution.

2.3.0.4 Data distribution:.

In the last substep, the data distributions have to be determined. A specific module M might be available in different versions using different data distributions usually resulting in different execution times. If we consider the execution of a module call M in isolation, the internal data distribution with the smallest execution time should always be used. This might however no longer be true, if we consider M in the context of other module calls that possibly require another data distribution for a specific variable. If we use the data distribution with the smallest execution time for M, a data redistribution before and after the execution of M might be necessary. The redistribution could possibly be avoided, if we use another version of M with a data distribution fitting to the data distributions of the surrounding modules. Thus, a smaller overall execution time could result although a suboptimal data distribution has been used for the call of M.

An analytical approach to compute optimal data distributions for arrays of arbitrary dimension is based on runtime formulas and parametrized data distributions that can be used to describe many regular distributions of arrays like blockwise or cyclic or block cyclic distributions [11]. The runtime functions contain additional parameters that describe the shape of the data distributions. An approach to select data distribution for basic modules in a coordination program is based on data distribution types and dynamic programming [15]. The composed modules are described by their corresponding module syntax tree. Bottom-up for each node of the module tree a time functions is determined that maps each data distribution type of the corresponding module M onto the minimal runtime that can be achieved for this data distribution type.

3. SPECIFICATION AND COORDINATION LANGUAGE

As described in Section 2 the specification program consists of a task parallel level comprising the hierarchical structure of CMs and a data parallel level containing BMs which are not decomposed further but express potential data parallelism.

The specification language for the task parallel level contains operators that describe how modules of an algorithm can be combined. The data parallel level consists of a set of C functions that perform internal communication with MPI operations. In the following, we concentrate on the task parallel level and describe the interaction with the data parallel level.

3.1 Specification language

A **specification program** consists of

- a list of external BM declarations,
- a list of external CM declarations, and
- a list of CM definitions.

An external *CM or BM declaration* has the form

modul_name(IN inputlist; OUT outputlist)

where modul_name denotes the CM or BM to be declared. The type and the arity of the module modul_name is given in form of a list of input parameters required by the module and the list of output parameters computed by the module. The inputlist of a module indicated by the keyword IN describes a number of input values x_i together with their types $type_i$, $i = 1, ..., m$, denoted as IN x_1 : $type_1, \dots, x_m : type_m$. Analogously, the outputlist indicated by the keyword OUT describes a number of output variables y_j with corresponding types $type'_j$, $j = 1, ..., n$, denoted by OUT y_1 : $type'_1, \dots, y_n : type'_n$. Data types include the types

scal , vec(n) , mat(n,n) , vec(n) $\rightarrow$ vec(n)

denoting scalars, vectors of size n, matrices of size $n \times n$, and multidimensional functions mapping vectors onto vectors.

The list of external BM declarations represents the interface to the lower data parallel level. The input and output parameters determine the data dependencies in the module structure. The actual code is given externally. The behavior of CMs is defined within the specification program by a module expression. External CMs are only declared in the specification program and are defined by module expressions in an external file. A *CM definition* is a CM declaration together with a definition of the behavior of the CM. It has the form

modul_name(IN inputlist; OUT outputlist) $=$ modul_expr,

where modul_expr is an expression built up from activations of other CMs and BMs declared or defined in the same specification program.

A *module expression* describes the hierarchical decomposition of a module into other BMs or CMs which are combined by appropriate constructors. An expression contains variable names for BMs (denoted by $\mathcal{B}_1, ..., \mathcal{B}_l$ in the following), variable names for CMs (denoted by $\mathcal{M}_1, ..., \mathcal{M}_k$), and variable names for data objects like scalars, vectors, matrices, or higher dimensional data objects (denoted by $x_1, ..., x_m$). Variables denoting BMs or CMs have a range and an image of fixed arity and type consistent with the arity and type in the declaration of that module. A module expression T is built up from activations of basic modules $\mathcal{B}(x_1, ..., x_m; y_1, ..., y_n)$ and activations of composed modules $\mathcal{M}(x_1, ..., x_m; y_1, ..., y_n)$, i.e., modules names together with a list of parameters corresponding to the declaration of the modules.

Module expressions are defined by the following grammar:

$$\begin{array}{rcl} T & ::= & \mathcal{B}(x_1, ..., x_m; y_1, ..., y_n) \\ & | & \mathcal{M}(x_1, ..., x_m; y_1, ..., y_n) \\ & | & T_1 \circ T_2 \\ & | & T_1 \parallel T_2 \\ & | & \text{for } (i = 0, ..., n)\ T \\ & | & \text{parfor } (i = 0, ..., n)\ T \\ & | & \text{while } (i, cond)\ T \\ & | & \text{if } (cond) \text{ then } T \\ & | & \{T\}. \end{array} \quad (1)$$

The meaning of the operators in a module expression T is the following: $\circ$ denotes a data dependent composition of modules, $||$ denotes a potential concurrent execution of modules, *for* and *while* denote sequential loops, *parfor* denotes a parallel loop, and *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. The operator $\circ$ denotes data dependency and not functional composition. Thus, the number of OUT parameters and IN parameters of two consecutive modules do not have to match. Modules combined by the $||$-operator may have the same input parameter but there is no interaction between the modules during their execution.

The actual input parameters, i.e., the parameters of a module call in a module expression can denote variables or constants; output parameters denote variables only. For input variables a call-by-value semantics is used, thus guaranteeing that the value given to a module on the left hand side are visible to all module expressions on the right hand side. For output variables a call-by-reference semantics is used so that the computed value of one module are visible to all succeeding modules in the same module expression.

The scope of visibility of data corresponds to the hierarchical structure of the module decomposition. So, input variables of a module activation in a module expression can include input variables of the module to be defined by that expression and all output variable of preceding module activations within the same module expression. Output variables of a modul activation in a module expression can be used as input variables for a following module activation within the same expression or as output variable of the entire module expression, i.e., as output of the module defined by this definition. The output of the last module activation in the expression should be part of the output of the module to be defined, as otherwise the result is lost and the computation would be redundant. Only output parameters of a module expression that are output of the module to be defined by that expression are visible to other parts of the module specification. All other output is visible only locally within the module expression.

The input parameter list can contain function parameters of type $vec(n) \to vec(n)$ mapping vectors onto vectors. Function parameters are included in the parameter list as many applications in scientific computing, such as solution methods for nonlinear equations or differential equations, are parametrized by a function describing the specific system to be solved. The use of function parameters is restricted to the evaluation of that function which is appropriate in that application area and avoids a complicated type system. As a consequence there are no function parameters as output parameters which is consistent with the discrete nature of numerical methods.

As example for a specification program we consider the Newton method in the next subsection.

3.2 Example: Newton iteration

The Newton method solves a nonlinear system of equations $F(z) = 0$ described by a nonlinear function $F : I\!R^n \to I\!R^n$. The Newton method iteratively computes a sequence of approximation vectors $z^0, z^1, \ldots$ for the solution $z \in I\!R^n$ according to the formula:

$$z^{(k+1)} = z^{(k)} - \left[DF(z^{(k)})\right]^{-1} F(z^{(k)}), \qquad i = 0, 1, 2, \ldots \quad ,$$

where $\left[DF(z^{(k)})\right] = \left(\frac{\partial F_i}{\partial z_j}(z^{(k)})\right)_{i,j=1,\ldots,n}$ denotes the Jacobian matrix of function F at $z^{(k)}$ and $z^{(0)}$ is an initial approximation. Each iteration step of the Newton method comprises several computations:

(1) The function F is evaluated at $z^{(k)}$.

(2) The entries of the matrix $\left[DF(z^{(k)})\right]$ are determined by a forward difference approximation.

(3) The vector $w^{(k)}$ is computed by solving the system of linear equations $\left[DF(z^{(k)})\right] w^{(k)} = -F(z^{(k)})$ by a direct method like the Gaussian elimination (instead of determining the inverse matrix of the Jacobian).

(4) The next approximation vector $z^{(k+1)} = z^{(k)} + w^{(k)}$ is computed.

(5) The error e $= ||w^{(k)}||$ is computed.

The Newton iteration ends if the error e is small enough and outputs the result of the last iteration k' as approximation solution $z^{app} = z^{(k')}$ of the system. Figure 3 illustrates the hierarchical decomposition of the Newton method into modules. Figure 4 shows the corresponding specification program. The decomposition starts with the module Newton and consists of four steps:

(I) The composed module Newton consists of a while-loop over NewtonBody which stops if e is small enough.

(II) NewtonBody activates ComputeCorrection computing a correction vector $w^{(k)}$ and ComputeIterate computing the new iteration vector $z^{(k+1)}$ and the error e.

(III) ComputeCorrection is decomposed into
  (1) EvaluateFunction for computing $F(z^{(k)}) = v$,
  (2) ComputeJaco for computing the Jacobian matrix DF, and
  (3) SolveLinSystem for solving the linear system providing the correction vector $w^{(k)}$.

  These steps are executed one after another which is expressed by the operation $\circ$.

(IV) ComputeIterate comprises the modules
  (4) Update for computing the next approximation $z^{(k+1)}$ and
  (5) ComputeError for computing the error e.

  These activations are independent from each other and can be executed in parallel which is expressed by the operator $||$.

(The numbers (1)–(5) correspond to the computation steps of the Newton method given above.) SolveLinSystem is an external module for solving a linear system of equations. Any appropriate solver can be used. ComputeJaco, Update, and ComputeError are data parallel BMs. EvaluateFunction is based on the specific function F describing the problem to be solved.

External BM Declarations:

```
  EvaluateFunction(IN F:vec(n)→vec(n), z:vec(n);
                   OUT v: vec(n))
  ComputeJaco(IN F:vec(n)→vec(n), z:vec(n), v:vec(n);
                   OUT DF: mat(n × n))
  Update(IN z: vec(n), w: vec(n); OUT znew: vec(n))
  ComputeError(IN w: vec(n); OUT e: scal)
```

External CM Declarations:

```
  SolveLinSystem(IN DF: mat(n × n), v: vec(n);
                   OUT w: vec(n))
```

Definitions:

```
  Newton(IN F:vec(n)→vec(n), z:vec(n), tol:scal, e:scal;
          OUT z: vec(n))
        = while( e < tol) NewtonBody(F, z; z, e)
  NewtonBody(IN F: vec(n) → vec(n), z: vec(n);
               OUT znew: vec(n), e: scal)
        = ComputeCorrection(F, z; w)
          ∘ ComputeIterate(w, z; znew, e)
  ComputeCorrection(IN F: vec(n) → vec(n), z: vec(n);
                     OUT w: vec(n))
          = EvaluateFunction(F, z; v)
            ∘ ComputeJaco(F, z, v; DF)
            ∘ SolveLinSystem(DF, v; w)
  ComputeIterate(IN w: vec(n), z: vec(n);
                  OUT znew: vec(n), e: scal)
          = Update(z, w; znew) || ComputeError(w; e)
```

Figure 4: Specification program of the Newton method.

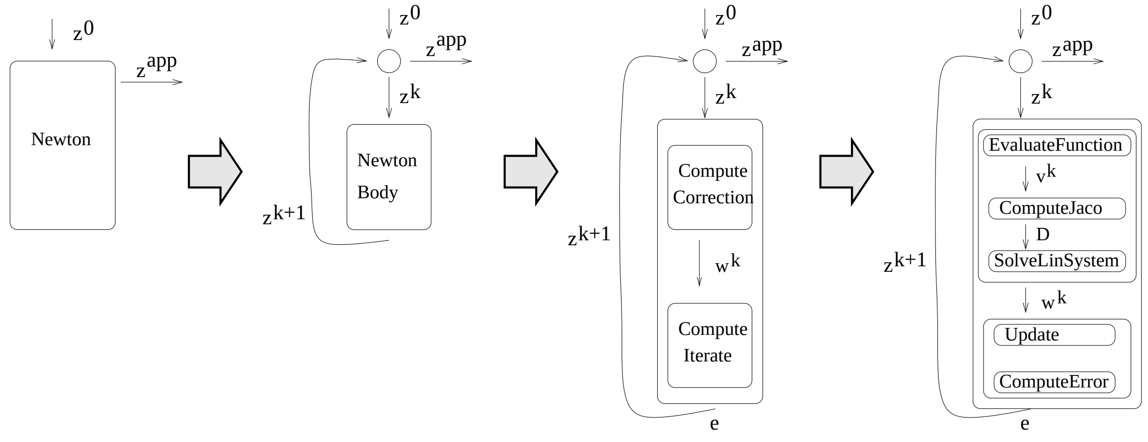


Figure 3: Decomposition steps for the Newton method resulting in a hierarchical module structure.

3.3 Coordination language

The coordination program is derived from the specification program in four substeps as described in Subsection 2.3. The corresponding design decisions are represented in the language by new constructors or by annotations.

3.3.0.5 Execution order and Pipelining.

Fixing the *execution order* is relevant for module activations that are combined by $||$ or parfor, since such modules can be computed either concurrently on subgroups of processors or one after another on the entire set of processors. The decision on the actual execution order is described by new operators which express the *execution order* (in contrast to the operators in the specification language where the data dependence between module activations is expressed). The binary operator $||$ of the specification language is either transformed into a *consecutive* execution expressed by the operator $';'$ or into a *concurrent* execution expressed by the same symbol $||$ as the original operator. Analogously, parfor is converted into seqfor or parfor expressing that a loop with independent iterations is realized consecutively or concurrently, respectively. It is also possible that a single parfor of the specification program is transformed into a double-nested loop of seqfor and parfor thus allowing a flexible adaptation to the number of processors available.

The operator $\circ$ of the specification language expresses a dependence between module activations. As described in Subsection 2.3 a pipelined realization of the module activations might be possible. We express pipelining in the coordination program by the operator $>$. A non-pipelined consecutive execution order is denoted by the same symbol $\circ$ as in the specification program.

In summary, the operators $||$ and $\circ$ of the specification language are transformed into the operators $||$, ; , $\circ$, and $>$ in the *coordination language*. Those operators exactly express all possible combinationss of data dependence and execution order as shown in the following table.

	data dependence	no data dependence
consecutive	$\circ$	;
concurrent	$>$	$\|\|$

To distinguish between a consecutive execution order with and without data dependence (i.e., using different operators $\circ$ and ;) is important for later decisions about the internal data distribution which may or may not cause expensive data redistributions.

3.3.0.6 Processor groups.

For the decision about the sizes of the *processor groups*, an additional parameter g is used for each module determining the number of executing processors. In the declarations of external BMs and CMs and in the definition of CMs, g is always the first parameter. The number of executing processors is determined for each module activation by using the parameter value of the calling module. For consecutive module activations, the same group sizes are used. For a concurrent execution of modules, each module is executed with a subset of the available processors.

3.3.0.7 Data distribution.

The distributions of the variables among the executing processors are described by *data distribution types*. Types provide a mechanism to express a complex data distribution as names so that the data distribution information can easily be passed to a called module or can be used to check whether data distribution types fit together. In addition to the usual data type, a *data distribution type* d is given for every input or output parameter of a module declaration, i.e., for a parameter x the entire information is $x : type : d$. For a module activation in a module expression, the data distribution information is implicitly passed to the called modules. If the distribution of the actual parameters do not fit to the distribution declared for the called module, a call to a redistribution module has to be inserted into the coordination program. If the same BM is available with different data distributions, these different versions are distinguished by different names for the corresponding functions.

3.3.0.8 Coordination program for the Newton method.

Figure 5 shows a coordination program to the specification program in Figure 4. The execution order in module ComputeIterate has been fixed to a consecutive execution. No pipelining is used. Using a Gaussian elimination for SolveLinSystem, a row-cyclic distribution of the Jacobian matrix DF is appropriate to get a good load balance. Correspondingly, the right hand side vector v will also be distributed cyclically, but the backward substitution delivers the output vector w replicated. This choice for SolveLinSystem leads to the remaining distributions shown in the figure, if redistributions are avoided.

4. EXAMPLES AND EXPERIMENTS

In this section, we consider some examples from numerical analysis for which specification and coordination programs have been

```
External BM Declarations:

  EvaluateFunction_cyclic(p; IN F: vec(n) → vec(n),
                                  z: vec(n): replic ;
                                OUT v: vec(n) : cyclic)
  ComputeJaco_rowcyclic(p; IN F: vec(n) → vec(n),
                                  z: vec(n): replic,
                                  v: vec(n): cyclic;
                                OUT DF: mat(n × n):rowcyclic)
  Update_replic(p; IN z: vec(n): replic, w: vec(n): replic;
                   OUT znew: vec(n): replic)
  ComputeError_replic(p; IN w: vec(n): replic;
                              OUT e: scal: replic)

External CM Declarations:

  SolveLinSystem_rowcyclic(p; IN DF: mat(n × n): rowcyclic,
                                   v: vec(n): cyclic;
                                 OUT w: vec(n): replic)

Definitions:

  Newton(p; IN F: vec(n) → vec(n), z: vec(n): replic,
               tol: scal: replic, e: scal: replic;
             OUT z: vec(n): replic)
        = while( e < tol) NewtonBody(p; F, z; z, e)
  NewtonBody(p; IN F: vec(n) → vec(n), z: vec(n): replic;
                 OUT znew: vec(n) :replic, e: scal :replic)
        = ComputeCorrection(p; F, z; w)
          ∘ ComputeIterate(p; w, z; znew, e)
  ComputeCorrection(p; IN F: vec(n) → vec(n),
                             z: vec(n): replic;
                           OUT w: vec(n): replic )
          = EvaluateFunction_cyclic(p; F, z; v)
            ∘ ComputeJaco_rowcyclic(p; F, z, v; DF)
            ∘ SolveLinSystem_rowcyclic(p; DF, v; w)
  ComputeIterate(p; IN w: vec(n): replic,
                          z: vec(n): replic;
                        OUTznew: vec(n): replic ,
                          e: scal: replic)
          = Update_replic(p; z, w; znew)
            ; ComputeError_replic(p; w; e)
```

Figure 5: Coordination program of a parallel Newton method for p processors.

developed according to the derivation process. We also give runtime experiments of the corresponding message-passing programs.

Runge-Kutta (RK) methods are one-step solution methods for initial value problems of ordinary differential equations (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 $y : I\!R \to I\!R^n$ is the unknown solution function and $f : I\!R \times I\!R^n \to I\!R^n$ is an application-specific function which is nonlinear in the general case. The vector $y_0 \in I\!R^n$ specifies the initial condition at x_0. The solution method computes approximation vectors y_κ for the exact values $y(x_\kappa)$ at discrete x-values $x_\kappa = x_{\kappa-1} + h$, $\kappa = 1, 2, ...$, one after another with *step-size* h. There is a large variety of RK methods that can be used for ODEs with different characteristics. In particular, we consider RK methods with a large potential of task and data parallelism. 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: (i) f has evaluation costs that are linear in the size of the ODE system (sparse function); (ii) f has evaluation costs that are quadratic in the size of the ODE system (dense function). In the experiments we used the Brusselator equation as example for a partial differential equation that results in a sparse system of ODEs and nonlinear partial differential equations solved with Fourier–Galerkin methods that results in a dense system of ODEs.

```
External BM Declarations:

  StageVector(IN f: scal × vec(n) → vec(n), x: scal,
                  y: vec(n), s: scal, A: mat(s × s),
                  h: scal, V: list[s] of vec(n);
               OUT vnew: vec(n))
  ComputeApprox(IN f: scal × vec(n) → vec(n), x: scal,
                    y: vec(n), s: scal, b: vec(s),
                    h: scal, V: list[s] of vec(n);
                 OUT ynew: vec(n))
  StepsizeControl(IN y: vec(n), ynew: vec(n);
                   OUT hnew: scal, xnew: scal)

Definitions:

  ItRKmethod(IN f: scal × vec(n) → vec(n), x: scal,
                 xend: scal, y: vec(n), s: scal,
                 A: mat(s × s), b: vec(s), h: scal;
              OUT X: list[] of scal, Y: list[] of vec(n) )
     = while( x < xend) {
           ItComputeStagevectors ( f, x, y, s, A, h ; V)
           ∘ ComputeApprox ( f, x, y, s, b, h, V ; ynew)
           ∘ StepsizeControl ( y, ynew; xnew, hnew) }
  ItComputeStagevectors(IN f: scal × vec(n) → vec(n),
                            x: scal, y: vec(n), s: scal,
                            A: mat(s × s),
                            h: scal;
                         OUT V: list[s] of vec(n))
     = InitializeStage(y ; V )
       ∘ for(j=1,...,m)
           parfor (l=1,...,s) StageVector(f, x, y, V ; Vnew)
```

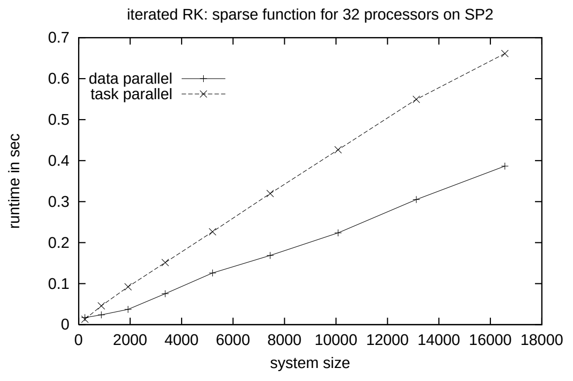
Figure 6: Specification program of the iterated RK method.

4.1 Iterated RK methods

Iterated RK methods are new solution methods for ODEs that have been derived from implicit RK methods for execution on parallel machines [18]. In each time step, an s-stage iterated RK method performs a fixed number m of iterations to compute s stage vectors $v^1, \ldots, v^s$ in each iteration step. The stage vectors computed in the last stage vector iteration m are used to compute the next approximation vector $y_{\kappa+1}$. The advantage of the iterated RK methods for parallel execution is that the iteration system of size $s \cdot n$ consists of s independent function evaluations that can be performed concurrently in a task parallel way.

Figure 6 shows the resulting specification program. In each iteration step of ItRKmethod, the stage vectors are computed in ItComputeStagevectors, the next approximation vector is computed in ComputeApprox, and the new step-size hnew and x-value xnew for the next iteration step are computed in StepsizeControl. The computation of the stage vectors is performed by a sequential loop with m iterations where each iteration is a parallel loop with s independent activations of the basic module StageVector.

In a coordination program, the parfor-loop over StageVector can be realized as group-parallel variant with mixed task and data parallelism or as pure data parallel variant. Experiments on several DMMs show that the pure data parallel variant performs better in most cases than a variant that exploits the available task parallelism, although the group-communication operations are usually faster than global communication operations. But a change from the group computation of the iteration vectors in StageVectors to the global computation of ComputeApprox requires additional (global) communication with costs that outweigh the savings by the group-communication. Figure 7 shows the resulting runtimes of a three-stage iterated RK method on an SP2 when applied to a sparse or dense ODE system, respectively.



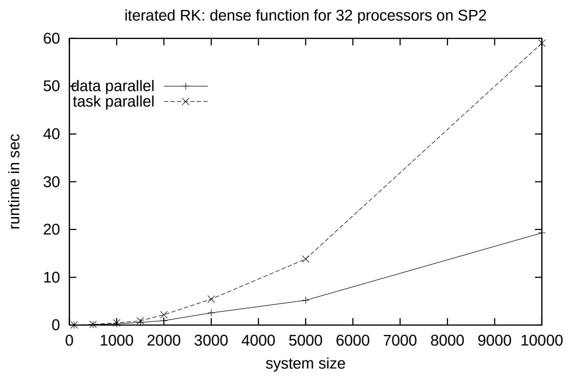
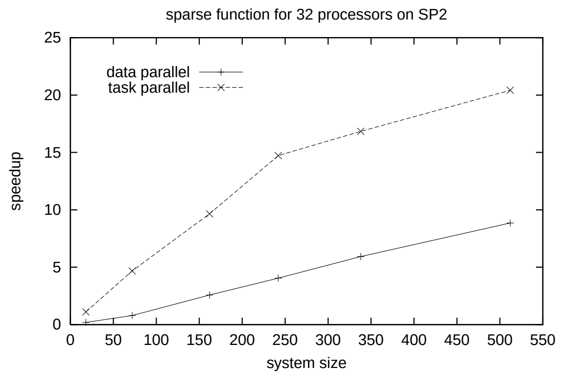


Figure 7: Runtimes in seconds on the IBM SP2: task parallel and data parallel execution schemes of the iterated RK method for sparse (top) and dense (bottom) ODE systems.



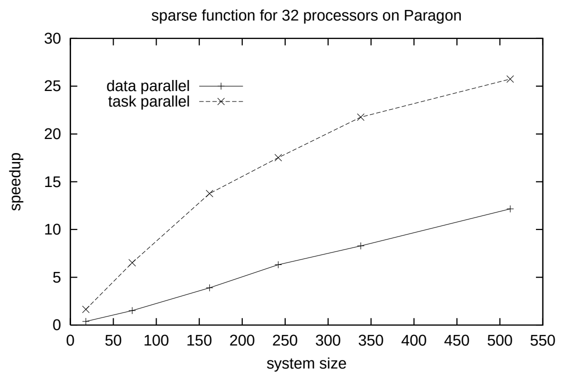


Figure 8: Speedups of task parallel and data parallel execution schemes of the DIIRK method on the IBM SP2 (top) and the Intel Paragon (bottom).

4.2 Diagonal-implicitly iterated RK methods

The iterated RK methods from Subsection 4.1 is an *explicit* RK method which is only suitable to solve *non-stiff* ODE systems. To provide an RK-solver for solving stiff ODE systems exhibiting task parallelism, the *diagonally iterated implicit RK* (DIIRK) method has been introduced. The computation scheme differs slightly from the iterated RK method. Functions evaluations are delayed and an additional diagonal matrix D is introduced. For a system of ODEs of size n, one stage vector iteration consists of s implicit nonlinear equations which are independent from each other.

The specification program for the DIIRK method is similar to the specification program for the iterated RK method in Figure 6, except that the computation of the stage vectors is different, i.e., the call StageVector(f, x, y, V ; Vnew) in the body of the parfor loop is replaced by the call Newton(F(f, x, y, s, A, h, l), Y, tol, e; Vnew) where Newton is the CM from Figure 4.

In the coordination program with task parallel execution of the DIIRK method, the s iteration vectors of one corrector step are computed by independent groups of processors where each group contains p/s processors. It is convenient to use the implementation of the Newton method from Figure 5. At the end of each corrector step, a redistribution has to be executed to make the iteration vectors available to all processors.

Implementations on different parallel machines (IBM SP2 and Intel iPSC/860 and Paragon) show that the group execution leads to a better performance than a consecutive execution order 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 in the Gaussian elimination are less expensive when executed in parallel on independent groups of processors (group broadcast). So, the combination of the modules in the specification program of the DIIRK method has the same structure as the iterated RK method but the transformation into a coordination program results in a different exploitation of the task parallelism.

Figure 8 shows the resulting speedup values on 32 processors of an IBM SP2 and Intel Paragon for an application of a DIIRK method with 5 stages to the solution of a stiff ODE system that results from a discretization of the Brusselator equation [8]. For dense ODE systems, the attained speedup values are larger for both the task parallel and data parallel execution. The task parallel execution is still much faster than the data parallel execution, but the difference in percentages is smaller.

4.3 Extrapolation methods

Extrapolation methods use a *generating method* like the Euler method and compute different approximations for $y_j(x_0 + \kappa H)$ with different step-sizes $h_0 < h_1 < \ldots < h_{r-1}$, e.g., defined by $h_j := H/n_j$ with $n_j := j + 1$, i.e., the generating method is applied n_j times for the computation of $y_j(x_0 + \kappa H)$. These approximations $y_j(x_0 + \kappa H)$ are then combined to obtain an approximation solution $y(x_0 + \kappa H)$ of higher order.

Extrapolation methods offer task parallelism since the computation of the different approximations with different step-sizes are independent from each other. In the specification program in Figure 9 this is expressed in the parfor-loop of MicroSteps where each activation of MicroSteps(j,...) computes one approximation $y_j(x_0 + \kappa H)$. The n_j steps of the generating method with the same step-size h_j have to be executed one after another. This is expressed in the definition of MicroSteps as a for-loop over i=1,...,j.

```
External BM Declarations:

   BuildExtrapTable(IN Y: list[r] of vec(n); y: OUT vec(n))
   ComputeMicroStepsize(IN j: scal, H:scal, r:scal;
                                  OUT $h_j$:scal)

External CM Declarations:

   EulerStep(IN f: scal $\times$ vec(n) $\rightarrow$ vec(n), x: scal,
                  y: vec(n), h: scal;
               OUT ynew: vec(n))

   StepsizeControl(IN y: vec(n), ynew: vec(n);
                     OUT hnew: scal, xnew: scal)

Definitions:

   ExtrapMethod(IN f: scal $\times$ vec(n) $\rightarrow$ vec(n), x: scal,
                      xend: scal, y: vec(n), r: scal, H: scal;
                   OUT X: list[] of scal, Y: list[] of vec(n) )
         = while( x $<$ xend)
               parfor(j=1,...,r) MicroSteps( j, f, x, y, H ; $y_j$)
               $\circ$ BuildExtrapTable(($y_1$,...,$y_r$) ; ynew)
               $\circ$ StepsizeControl(y, ynew ; Hnew, xnew)
   MicroSteps(IN j:scal , f: scal $\times$ vec(n) $\rightarrow$ vec(n), x: scal,
                 y: vec(n), H: scal;
              OUT ynew: vec(n))
         = ComputeMicroStepsize(j, H, r ; $h_j$)
            $\circ$ for(i=1,...,j) EulerStep(f, x, y, $h_j$ ; ynew)
```

Figure 9: Specification program of the extrapolation method.

The generating method provides additional data parallelism which can be exploited by distributing the components of the approximation vectors among the processors.

The difference concerning a parallel implementation between the iterated RK methods from the last subsection and extrapolation methods is that for extrapolation methods each of the independent computations require a different amount of computational work. Hence, for a task-parallel implementation, an appropriate load balancing scheme should be used to guarantee that the different approximations are available at about the same time so that a small parallel runtime results.

In a concurrent execution order using r disjoint processor groups the size of the groups can be adapted to the computational work, i.e., group j obtains about $g_j = p \cdot n_j / \sum_{i=1}^{r} n_i$ processors. This design decision is expressed in a coordination program by annotating the group information to the body of the parfor-loop. Another possibility is to use $\lceil r/2 \rceil$ disjoint processor groups containing the same number of processors and each group performs the generating method with two different step-sizes h_j and h_{r-j}. This scheduling and group-size decision is expressed in the coordination program by transforming the parfor-loop into a mixed parfor- and seqfor-loop over module actications of MicroSteps annotated with the same group size.

5. RELATED WORK

A large variety of programming models and languages have been proposed with different levels of abstraction. A good overview of related work on programming paradigms can be found in [9, 16]. The most important language approaches include functional programming languages and algorithmic skeletons for which parallelism is implicitly available as independent expressions can be evaluated in parallel, and data parallel languages like HPF (High Performance Fortran) [5] or NESL [3].

Other language approaches include Braid, Fortran M, Fx, Opus, and Orca [1]. Fortran M [7] allows the creation of processes which can communicate with each other by predefined channels and which can be combined with HPF Fortran for a mixed task and data parallel execution.

The Fx model expresses task parallelism by providing declaration directives to partition processors into subgroups and execution directives to assign computations to different subgroups (task regions) [17]. Task regions can be dynamically nested, i.e., a procedure call made in a task region can further subdivide the executing processors using another task region directive. A model that is similar to the task parallelism model of Fx has recently been added to High Performance Fortran [5] as an approved extension. Although the Fx approach is similar in spirit to our approach, there are some important differences. The task regions in Fx cannot be nested lexically whereas our model 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 our model requires all task coordination to be performed on the upper level of the program derivation process. 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. Our 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, our model provides a framework for the complete derivation process in which support tools can be integrated quite naturally.

An exploitation of task and data parallelism in the context of a parallelizing compiler can be found in the Paradigm compiler [2] which provides a framework that expresses task parallelism by a macro dataflow graph derived from the hierarchical task graphs used in the Parafrase compiler. Nodes in the macro dataflow 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.

6. CONCLUSIONS

We have presented a new model with an integrated language support to derive efficient parallel implementations. Important features of the model are

- an exploitation of data parallelism and arbitrary levels of task parallelism,
- a structured derivation of parallel implementations that are efficient for a specific parallel machine, and
- a clear separation and interface between the data parallel and the task parallel executions.

The model enables the programmer to derive efficient implementations in well-defined steps where each step concentrates on one design decision. Thus, the model provides an easy-to-use framework for the programmer and is the basis for an automatic or semi-automatic derivation of implementations from a general machine-independent specification. Currently, a prototype is available that allows the generation of MPI programs from coordination programs [4]. Moreover, optimization algorithms have been developed and implemented to support the derivation steps. Future research includes the integration of these algorithms into an integrated tool to support the programmer in the derivation of parallel implementations.

7. REFERENCES

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

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

[3] G.E. Blelloch. Programming Parallel Algorithms. *Communications of the ACM*, 29(3):85–97, March 1996.

[4] U. Fissgus, T. Rauber, and G. Rünger. A framework for generating task parallel programs. In *7th Symposium on the Frontiers of Massively Parallel Computation - Frontiers '99*, pages 72–80, Annapolis, Maryland, 1999.

[5] High Performance Fortran Forum. High Performance Fortran Language Specification. *Scientific Programming*, 2(1), 1993.

[6] The MPI Forum. MPI: A Message Passing Interface Standard. Technical report, University Tennessee, April 1994.

[7] I. Foster and K.M. Chandy. Fortran M: A Language for Modular Parallel Programming. *Journal of Parallel and Distributed Computing*, 25(1):24–35, April 1995.

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

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

[10] 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*, 8(11):1098–1116, 1997.

[11] T. Rauber and G. Rünger. Optimal Data Distribution for LU Decomposition. In *Proc. of the EuroPar'95*, Springer LNCS 966, pages 391–402, 1995.

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

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

[14] T. Rauber and G. Rünger. Compiler Support for Task Scheduling in Hierarchical Execution Models. *Journal of Systems Architecture*, 45:483–503, 1998.

[15] T. Rauber, G. Rünger, and R. Wilhelm. Deriving Optimal Data Distributions for Group Parallel Numerical Algorithms. In *Proc. 2nd Conf. on Massively Parallel Programming Models*, pages 33–41, Berlin, Germany, 1995.

[16] D. Skillicorn and D. Talia. Models and languages for parallel computation. *ACM Computing Surveys*, 30(2):123–169, 1998.

[17] J. Subhlok and B. Yang. A New Model for Integrating Nested Task and Data Parallel Programming. In *8th ACM SIGPLAN Symp. on Principles & Practice of Parallel Programming*, pages 1–12, 1997.

[18] 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.

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