Communicating Multiprocessor-Tasks

JÖRG DÜMMLER
Chemnitz University of Technology
djo@informatik.tu–chemnitz.de

THOMAS RAUBER
University Bayreuth
rauber@uni–bayreuth.de

GUDULA RÜNGER
Chemnitz University of Technology
ruenger@informatik.tu–chemnitz.de

Abstract

The use of multiprocessor tasks (M-tasks) has been shown to be successful for mixed task and data parallel implementations of algorithms from scientific computing. The approach often leads to an increase of scalability compared to a pure data parallel implementation, but restricts the data exchange between M-tasks to the beginning or the end of their execution, expressing data or control dependencies between M-tasks.

In this article, we propose an extension of the M-task model to communicating M-tasks (CM-tasks) which allows communication between M-tasks during their execution. In particular, we present and discuss the CM-task programming model, programming support for designing CM-task programs, and experimental results. Internally, a CM-task comprises communication and computation phases. The communication between different CM-tasks can exploit optimized communication patterns for the data exchange between CM-tasks, e.g., by using orthogonal realizations of the communication. This can be used to further increase the scalability of many applications, including time-stepping methods which use a similar task structure for each time step. This is demonstrated for solution methods for ordinary differential equations.

1 Introduction

The implementation of modular programs on parallel platforms can be supported by multiprocessor task programming (M-task programming). Each M-task represents a part of a program which can be executed in parallel by an arbitrary number of processors. The entire program consists of a set of M-tasks; a coordination structure specifies how the M-tasks of one specific program cooperate with each other and which dependencies have to be considered for the execution. For the coordination of M-tasks different parallel programming models have been proposed [13, 14, 15, 21]. A coordination structure in form of SP-graphs (serial parallel graphs) has been used in the TwoL model [15]. Using M-tasks often leads to a better scalability compared to a pure data parallel implementation due to a decrease of the communication overhead. Executing M-tasks concurrently on smaller subsets of processors reduces the internal overhead

for collective communication of the M-tasks, thus reducing the overall communication overhead.

An M-task can use data produced by another M-task, leading to dependencies between M-tasks that have to be considered for their execution. A dependency between two M-tasks may require communication to achieve a data re-distribution such that a data structure is reordered at the end of one M-task A to be available in a data distribution expected by another M-task B before the execution of B starts. This restricts the data exchange between M-tasks to the beginning or the end of their execution.

In this article, we extend the standard M-task model as used in the TwoL model to the model of communicating M-tasks (CM-tasks) which allows a more complex graph structure and an additional kind of communication between M-tasks. The extension includes modified M-tasks which have the ability to communicate with other M-tasks during their execution. This new feature can capture the behavior of applications from scientific computing or numerical analysis in which modules exchange information during their execution. Examples are modules with internal iterations exchanging data with other modules after each iteration step. The CM-task model can also benefit from specific communication patterns. For example, it is possible to organize the communication phases between CM-tasks in an orthogonal fashion, thus enabling a more efficient realization of array-based applications on many execution platforms.

The CM-task programming model requires new scheduling and load balancing algorithms to achieve an efficient execution. The scheduling has to ensure that CM-tasks which communicate with each other are executed concurrently to each other on disjoint sets of processors. The scheduling has to be based on a cost model which also takes the internal computations and the external communications between CM-tasks into consideration. To support the programming in the CM-task model we have designed a transformation framework including a specification mechanism for CM-task programs and transformation steps which create an executable parallel program.

In the following, we present the parallel programming model of CM-tasks in Section 2 and discuss the programming support in Section 3. As example applications, we consider parallel Adams methods [16] which are solvers for systems of ordinary differential equations (ODEs) with potential method parallelism and show experimental results in Section 4. Section 5 discusses related work and Section 6 concludes.

2 Programming model of CM-tasks

The CM-task programming model exhibits two levels of parallelism: an upper level that captures the coarse-grain task structure of the application and a lower level that expresses parallelism within the tasks of the upper level. A CM-task program consists of a collection of CM-tasks which form the tasks of the upper level. Each CM-task is implemented in a way that allows its execution on an arbitrary number of processors. A CM-task can be a parallel module performing parallel computations (basic CM-task) or can have an internal structure activating other CM-tasks (composed CM-task). The internal parallelism of basic CM-tasks is realized by an SPMD programming approach; message passing may be used for distributed memory platforms while an implementation based on Pthreads or OpenMP may be advantageous on clusters with large SMP

nodes. But within one CM-task program, the same SPMD model for the basic CM-tasks is used. In this article, we assume that CM-tasks are based on message passing using MPI and have an internal data distribution for each of their input and output variables. On the upper level, the CM-tasks of the same parallel program can cooperate with each other in two different ways:

1) **P-relation**: CM-tasks A and B have a precedence relation (P-relation) if CM-task B requires input data from CM-task A before it can start its execution. This relation is not symmetric and is denoted by $A\delta_P B$.
2) **C-relation**: CM-tasks A and B have a communication relation (C-relation) if A and B have to exchange data during their execution to be able to continue their execution correctly. This relation is symmetric and is denoted by $A\delta_C B$.

In contrast, previous programming models based on M-tasks allow only P-relations between the tasks. The P- and C-relations determine some constraints on the potential execution order of CM-tasks:

- If there is a P-relation between two CM-tasks A and B, they have to be executed one after another. If B expects its input data in another data distribution as it is produced by A, a re-distribution operation has to be used to make the data available in the distribution expected. This re-distribution has to capture the situation that the processor sets executing A and B are not identical and may even be disjoint.
- If two CM-tasks A and B have a C-relation, both tasks have to be executed concurrently to realize the specified data exchange during their execution. Therefore, A and B are executed on disjoint sets of processors and cannot be executed one after another.
- Due to the constraints on CM-tasks with C-relations to be executed at the same time and for CM-tasks with P-relation to be executed one after another, there cannot be both a P-relation and a C-relation between two CM-tasks.
- If there is no P-relation and no C-relation between two CM-tasks A and B, they can be executed concurrently to each other but also one after another.

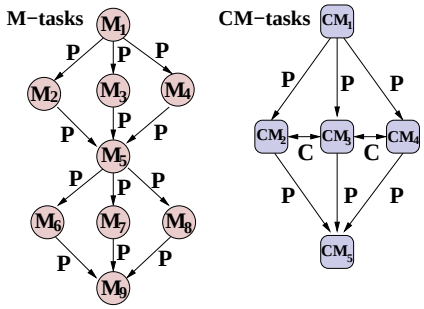

Figure 1: M-task graph (left) with P-relations and CM-task graph (right) with P- and C-relations.

A CM-task program can be represented as a CM-task graph $G = (V, E)$ where the set of nodes $V = \{A_1, \ldots, A_n\}$ represent the CM-tasks. The edges are composed of two sets $E = E_c \cup E_p$ with $E_c \cap E_p = \emptyset$; E_p contains directed edges and represents the P-relations between CM-tasks; E_c contains bidirectional edges and represents the C-relations between CM-tasks.

Figure 1 illustrates an M-task graph (left) and a CM-task graph (right) for a typical task graph structure occurring in time stepping methods, e.g. for the solution of ODEs. The M-task graph captures two time steps where M-tasks M_2, M_3, and M_4 perform independent computations within one time step and the tasks M_6, M_7, and M_8 perform analogous computa-

tions for the next time step. In between, M_5 combines the results, e.g. for error control or information exchange. In the M-task model (with P-relations only), M_2 and M_6 cannot be combined because the result of M_2 is used by M_5. In the CM-task model, such combinations are possible, see Figure 1 (right). The CM-tasks CM_2, CM_3, and CM_4 are used to perform the independent computations within a series of time steps and to combine the results at the end of each time step. Data exchanges with other program parts are captured by C-relations.

The CM-task graph of a CM-task program illustrates constraints on the execution order. Different execution orders are possible, but will usually result in different execution times. The goal is to find a schedule and mapping for the CM-tasks of one program which fulfills the constraints given by the CM-task graph and leads to a minimum execution time on a given parallel execution platform.

3 Programming Support

To support the development of CM-task programs, a specification language, a cost model, and a transformation framework with support tools have been developed.

3.1 Specification language

The specification language is used to describe the upper-level of CM-task programs by giving a list of CM-task declarations. The dependencies (P-relations) and interactions (C-relations) between CM-tasks are expressed by variables which carry the information to be communicated. For a P-relation between CM-tasks A and B, specific variables are produced by A as output data and are required by B as input data. For a C-relation between CM-tasks A and B, specific variables are exchanged between A and B or are sent from A to B (or from B to A) during the execution of A and B.

A CM-task specification of an application consists of data type declarations, data distribution type declarations, declarations of CM-tasks supplied by the user (basic CM-tasks), and definitions of CM-task graphs (composed CM-tasks). As data types we consider scalars and multi-dimensional array structures. For the data distribution, arbitrary block-cyclic and replicated distributions over multi-dimensional processor meshes are available. The specification contains only the interface definition of the CM-tasks. The implementation of the basic CM-tasks are provided separately by the programmer using the corresponding data distribution.

The declaration of a basic CM-task starts with the keyword **cmtask** followed by a unique name and two parameter lists: an input/output parameter list in round brackets for variables that are communicated over the P-relations at the beginning or the end of the CM-task and a communication parameter list in square brackets for variables that are exchanged during the execution of the CM-task. Each parameter has a name and a data type. The input/output parameters additionally have an access type (in, out, inout) and array variables have a data distribution type. An estimation of the execution time based on the cost model, see Subsection 3.2, can also be specified.

Composed CM-tasks are defined by using the keyword **cmgraph** followed by the name and the input/output parameter list similar to the parameters for basic CM-tasks.

Listing 1: Specification program for the PAB method.

```
const K=8, n=320000;
type vector = array[n] of double;
distrib vector:replic = [REPLIC(p)];
cmtask pab_stage (stage:int:in, xs,xe,h:double:inout,
  yps:vector:inout:replic)[xchg:vector] with runtime
  n/p*T_eval+(2*K+1)*n/p*T_op+T_mb(p, n/p);
cmmain pab (xs,xe,h:double:in, yps:vector[K]:inout:replic) {
  var vecxchg : vector;
  parfor (i = 0:K-1) {
    pab_stage (i, xs, xe, h, yps[i])[vecxchg]; } }
```

One composed CM-task is defined as the main entrance point of the CM-task program; this CM-task is denoted by using the keyword **cmmain** instead of **cmgraph**. The body of composed CM-tasks may include the declaration of local variables using the keyword **var**. Loops and conditional statements are available to define the internal task structure of composed CM-tasks. Different types of loop structures are supported: sequential **for** and **while**-loops can be used to define the sequential execution of CM-tasks. Parallel **parfor**-loops can be used to activate a set of CM-tasks that are executed concurrently on disjoint subsets of processors. The iteration space of the **for** and **parfor**-loops has to be known at compile time (constant loop bounds) whereas the **while**-loop contains an estimation of executed iterations. Conditionals are expressed by using the keyword **if** and may contain an optional **else** branch.

The activation of a CM-task is specified by giving the name of the CM-task, an input/output parameter list (for the P-relations), and a communication parameter list (for the C-relations). The P-relations and C-relations of a CM-task graph are defined implicitly by using variable names in the parameter lists. The transformation steps of the framework annotate additional information to the composed CM-task definitions including the explicit specification of the relations, scheduling and load balancing decisions, and information about necessary data re-distribution operations; see Subsection 3.3 for more details.

Example As an example for scientific applications that can benefit from the CM-task programming model we consider parallel Adams methods which are solution methods for ordinary differential equations (ODEs). These methods have been developed for a parallel implementation in [20] and include the explicit parallel Adams-Bashforth (PAB) methods as well as the implicit parallel Adams-Moulton (PAM) methods. Combining the PAB method with the PAM method in a predictor-corrector scheme results in an implicit ODE solver (PABM) with fixed point iteration using the PAB method as predictor. In [16], a detailed description of a parallel implementation is given.

Both, the PAB and PABM methods compute a fixed number K of stage vectors in each time step which are then combined to compute the final solution vector of the time step. In the M-task model, the stage vectors of one time step can be computed by separate M-tasks which are executed concurrently by disjoint sets of processors. This has the advantage that the internal communication of the M-tasks (which is dominated

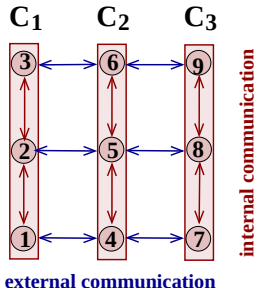

Figure 2: Orthogonal communication between CM-tasks: Processor subsets C_i with $C_1 = \{1, 2, 3\}$, $C_2 = \{4, 5, 6\}$ and $C_3 = \{7, 8, 9\}$ are used for executing CM-task CM_i, $i = 1, 2, 3$. Orthogonal communication for communication between CM-tasks is performed within the subsets $\{1, 4, 7\}$, $\{2, 5, 8\}$, and $\{3, 6, 9\}$.

by a gather operation, e.g. MPI_Allgatherv()) is restricted to a subset of the processors. At the end of each time step, global communication is required to construct the solution vector of the time step. For x time steps, the total number of M-tasks is $x \cdot K$. Using the CM-task model, it is now possible to define CM-tasks such that one CM-task is responsible for the computation of the corresponding stage vectors in all x consecutive time steps, i.e. a total number of K CM-tasks is used, independently from the number of time steps. This enables the use of orthogonal communication between the CM-tasks at the end of each time step to construct the solution vector of the time step. For many array-based algorithms from scientific computing with potential CM-task parallelism, this can reduce the communication overhead tremendously.

The term orthogonal communication denotes a communication pattern for processors arranged in a two-dimensional mesh structure and divided in two different ways into subsets of processors with corresponding communicators. The first division into subsets of processors $C_1, \ldots, C_K$ is used to execute CM-tasks $CM_1, \ldots, CM_K$ in parallel, each one executing one CM-task. The internal communication of CM-task CM_i is executed within subset C_i, $i = 1, \ldots K$. The second division into subsets results by building new subsets across the subsets $C_1, \ldots, C_K$; these orthogonal sets of processors contain one processor of each of the subsets $C_1, \ldots, C_K$ and are used for the communication between concurrently running CM-tasks, see Figure 2 for an illustration. In the example, the second communicator is used for the data exchange after each time step using a multi-broadcast operation and includes all processors with the same rank within the first communicator.

Listing 1 shows the specification program for the PAB method with $K = 8$ stage vectors for an ODE of size $n = 320000$. For the replicated storage of the stage vectors a data type `vector` and a distribution type `replic` (for replicated distribution) are declared. The CM-task that computes the stage vectors is called `pab_stage` and requires the stage number `stage`, the starting time `xs`, the ending time `xe`, and the step size `h` as an input. The parameter `yps` inputs the initial stage vector and outputs the final result after all time steps have been computed. The communication parameter `xchg` is used to exchange information with the CM-tasks computing the other stage vectors after each time step. The cost information provided is discussed in Subsection 3.2. The composed CM-task `pab` is the main part of the application. It consists of a parallel loop that creates K independent CM-tasks `pab_stage`. Because all loop iterations access the same local variable `vecxchg`, there is an implicit C-relation between each pair of iterations.

3.2 Cost model

The specification language is embedded into a compiler framework which supports design decisions for the parallel execution on a specific execution platform, like the execution order of independent CM-tasks, assigning processors to CM-tasks, and determining required data re-distributions between cooperating CM-tasks. The design decisions are based on estimated costs for the execution of CM-tasks and the communication between them. Usually, different execution orders are possible for a given specification program, and each possible execution order may result in different estimated costs. The compiler framework selects the execution with the smallest estimated costs for the execution platform considered.

The cost model is based on symbolic runtime formulas which estimate the expected execution time of CM-tasks for a specific set of processors on the given machine and for a specific size of the input data. The cost model captures the expected execution times of the basic CM-tasks and the communication costs resulting from data re-distribution operations induced by the P-relations. The costs for a basic CM-task consist of computation costs for the arithmetic operations and communication costs for internal communication; also costs for data exchanges as specified by the C-relations are considered. The data re-distribution costs depend on the size of transmitted data in bytes and on the platform dependent startup time and byte-transfer time; the size of transmitted data can be computed within the framework based on the data types and data distribution types. Costs for composed CM-tasks can be built up from costs of basic CM-tasks and communication times for P-relations and C-relations according to the hierarchical CM-task structure: For a concurrent execution of CM-tasks CM_1 and CM_2, the maximum of their cost formulas is taken; for a consecutive execution, the sum of their cost formulas is used. The costs for the CM-task **cmmain** determine the costs for the entire program.

The symbolic runtime formulas are based on application dependent information and platform dependent information. The application dependent information includes the number of arithmetic operations and the number and types of communication operations. The platform dependent information includes the average execution time for an arithmetic operation and formulas describing the execution time for the communication operations depending on the number of transmitted data items and the number of participating processors. The cost information is included in the CM-task specification and can be provided manually by the programmer if simple cost formulas are used or can be extracted automatically by a compiler tool by inspecting the internal SPMD structure of the CM-task implementations.

In [11] it has been shown that symbolic runtime formulas can give realistic predictions of the runtime of the PAB and the PABM method. For the CM-task `pab_stage` of the PAB method the cost formula $T_{pabstage}(n,p) = (n/p * T\_eval + (2*K+1) * n/p * T\_op) + T\_mb(p, n/p)$ has been derived, see Listing 1. In this formula, K represents the number of stage vectors, n is the size of the ODE system, p is the number of processors, T_eval is the time to evaluate a single component of the ODE system, T_op is the time to execute an arithmetic operation and T_mb is the runtime of a multi-broadcast operation (MPI_Allgatherv()) depending on the number of processors and the size of the data. All values, except p, are known at compile-time. This results in

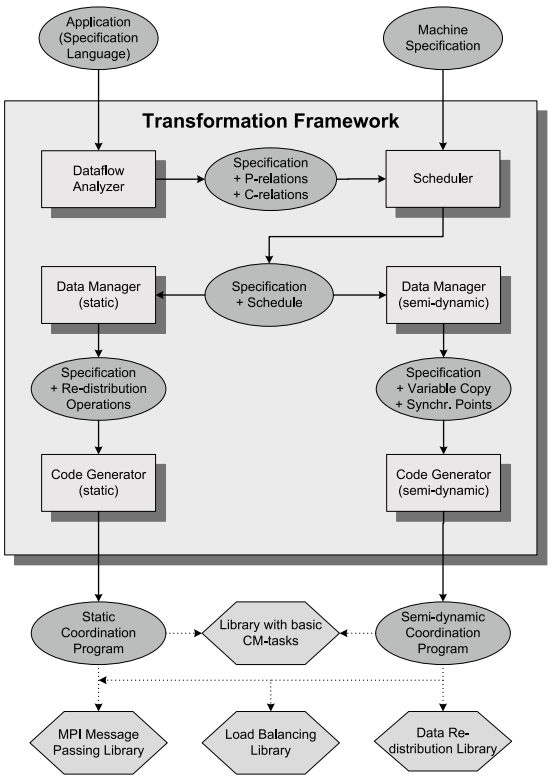

Figure 3: Overview of the transformation framework.

the cost formula $K \cdot T_{pabstage}(n, p)$ for one time step of a data parallel version of the PAB method executing all stage vectors one after another by all processors.

3.3 Transformation Framework

A compiler framework is provided to transform CM-task programs specified in the specification language into executable parallel MPI programs. The framework integrates scheduling and load balancing methods, data distribution methods, as well as a generation process for the final MPI program. The framework supports two different approaches to generate parallel programs:

- The **static approach** of the framework generates an MPI program (in C) with a fixed schedule, i.e. the execution order of the CM-tasks and the size of the processor groups used for the execution is fixed at compile time and cannot be changed at runtime. The fixed schedule is created for a given problem instance (e.g. a fixed system size) and a specific target platform with a fixed number of processors. This approach is especially suited for dedicated homogeneous platforms and requires an accurate cost model for a good schedule.
- The **semi-dynamic approach** of the framework generates an MPI program (in C)

with an initial plan for an execution order of the CM-tasks and an initial size of the processor groups used for the execution. This initial plan is based on a fixed schedule for a default problem instance and a default target platform. The MPI program generated allows the integration of a load balancing module that is able to arrange dynamic reorganizations of the processor groups executing CM-tasks based on observations of the dynamic behavior of the execution progress and possible load imbalances. Thus, semi-dynamic programs are able to adapt to different problem instances and varying target platforms, i.e., they make use of additional processors, if available, and compensate for load imbalances resulting from platform heterogeneity or an uneven distribution of workload. This approach is especially suited for non-dedicated heterogeneous platforms.

The input to the framework consists of (a) a description of the CM-task application in the specification language and (b) the platform dependent part of the cost information in a separate *machine specification*. The generated program uses implementations of basic CM-tasks that are provided by the programmer as parallel MPI functions. The interface of each of these MPI functions has to match the specification, i.e., the number and types of the parameters have to match; the data distribution types are used to select appropriate re-distribution operations. At runtime, the generated program provides two kinds of communicators to the basic CM-tasks: (a) a group communicator for group internal communication and (b) a cluster communicator that includes all processes that execute CM-tasks that are interconnected by C-relations for communication between running CM-tasks.

The programs generated by the semi-dynamic approach additionally use a load balancing library and a data re-distribution library. The load balancing library is initialized at program start with the CM-task graph of the application and is invoked during the execution of the application with measured runtimes of executed CM-tasks and may output an adapted schedule. The data re-distribution library provides runtime support for copying and re-distributing data structures.

The transformation framework includes a number of transformation steps where each step generates new information and adds it to the application description. Additionally, support tools are provided to visualize the progress of the framework and to give the programmer a possibility to interact with the framework, e.g., to influence or change decisions made by the framework. Figure 3 gives an overview of the transformation system. In the following, we describe the transformation steps in more detail.

The **Dataflow Analyzer** uses a data dependency analysis to detect the P-relations and C-relations that are defined implicitly in the initial specification program. For the P-relations, three different kinds of data dependencies are considered between the input/output parameter lists of the CM-tasks forming a CM-task graph: a WR data dependency occurs when a CM-task A writes a variable that is subsequently read by a CM-task B; a RW data dependency emerges when a CM-task A reads a variable that is subsequently written by a CM-task B; a WW data dependency arises when CM-tasks A and B subsequently write to the same variable. In each of these cases a P-relation between CM-tasks A and B is inserted; for WR data dependencies this P-relation is additionally annotated with the name of the variable, denoting that a data re-distribution between A and B might be necessary.

The C-relations of a CM-task graph are constructed using an analysis of the communication parameter lists of the CM-tasks. Two cases are considered: (a) two CM-tasks A and B access the same communication variable denoting a point-to-point communication between A and B during their execution and therefore a single C-relation is created; (b) more than two CM-tasks access the same communication variable resulting in collective communication between these CM-tasks and therefore C-relations between each pair of these CM-tasks are inserted.

The **Scheduler** determines a global hierarchical schedule consisting of a starting point in time and an executing processor group for each CM-task in a given specification of a CM-task application. Heuristics or hand-coded scheduling can be used for the scheduling decisions.

The **Static Data Manager** inserts descriptions of data re-distribution operations into the specification language. Such a description consists of the starting point in time, the source and target processor groups and a list of variables that should be re-distributed. For each variable, the name, the data type and the source and target distribution type is specified. The required data re-distribution operations are determined by an inspection of the P-relations within each composed CM-task.

The **Static Code Generator** produces a static coordination program that utilizes the MPI message passing library for the processor group management and for the realization of the data re-distribution operations. The coordination program consists of an initialization phase that creates all required communicators, a coordination function for each composed CM-task, and a finalization procedure that disposes all created communicators. A coordination function may contain declarations of local variables, constructs to guide the control flow (if-statement, for-loop) and code to execute CM-tasks and data re-distribution operations. The data re-distribution operations are performed in three steps: first, all sending processors pack their data into a sending buffer; second, the data is transmitted over the network; and third, the receiving processors unpack the data into the appropriate memory locations.

The **Semi-dynamic Data Manager** contributes to the transformation process in two ways. First, it marks the positions in the specification program where the load balancing should be performed. By default, the marked positions are points in time where all processors are available to allow a global restructuring and within loops to allow an adaption of the schedule based on previous loop iterations. Second, this transformation step decides which variable accesses are performed to the original variable and for which accesses a copy of the original variable should be supplied. The original variable may only be accessed by at most one CM-task at any given point in time. Write accesses use the original variable to ensure that it always contains the most recent values. This approach provides a flexible way to deal with a changing processor group layout without having to recompute all required re-distribution operations at runtime.

The **Semi-dynamic Code Generator** produces a coordination program that consists of a coordination function for each composed CM-task. Before starting a CM-task the required communicators are created and the data re-distribution library is invoked to ensure a correct data distribution of the input data. The runtimes of the executed CM-tasks are measured and provided to the load balancing library at the positions marked by the previous transformation step.

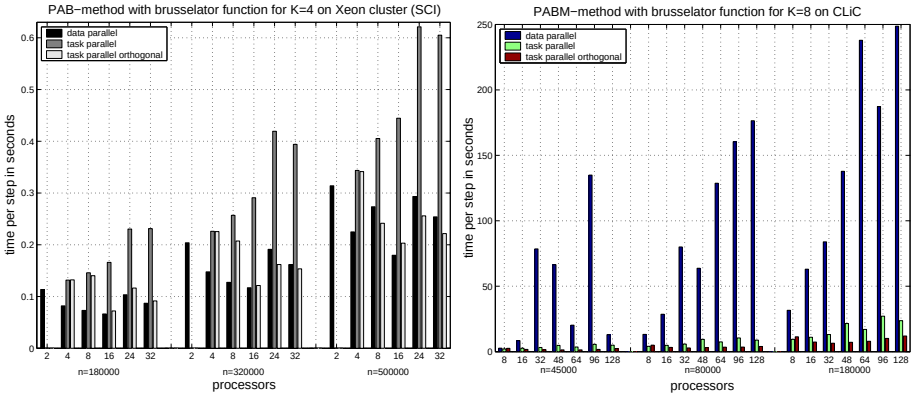

Figure 4: Runtimes of one time step of the PAB method for Brusselator on SCI Xeon cluster with $K = 4$ (left) and runtime of the PABM method on CLiC with $K = 8$ (right).

4 Experimental evaluation

In the following, we illustrate the CM-task model for solution methods of ODEs. In particular, we consider the PAB and PABM methods that have been introduced as examples in Subsection 3.1. For the runtime tests we consider three different program versions using a static schedule:

- The pure data parallel version computes the stage vectors one after another using all available processors. Communication between the different stage vector computations is not required.
- The task parallel version uses K disjoint processor groups of equal size to compute the K stage vectors in parallel. Internally, each task is executed in an SPMD fashion resulting in mixed task and data parallelism for the entire program. Additional communication operations are required at the end of each time step to exchange the stage vectors. This communication is realized by an intra group broadcast followed by an inter group data exchange.
- The orthogonal version uses the same task layout as the task parallel variant. The exchange of stage vectors is performed using concurrent multi-broadcast operations between processes with the same group rank.

The runtime tests shown are made for ODE systems that result from a spatial discretization of the 2D Brusselator equation [7]. The resulting ODE systems are sparse: each component of the right-hand side function **f** of the ODE system has a fixed evaluation time that is independent of the size of the ODE system; thus, the evaluation time for the entire function **f** increases linearly with the size of the ODE system. The figures show the execution time of one time step, obtained by dividing the total execution time by the number of time steps performed. A typical integration may consist of tens of thousands of time steps, thus leading to a large overall execution time.

Figure 4 (left) shows the runtimes for a Xeon cluster consisting of 16 dual SMP nodes with an SCI interconnection network using ScaMPI. For two processors, no task parallel implementation is given because at least $K = 4$ processors are required for

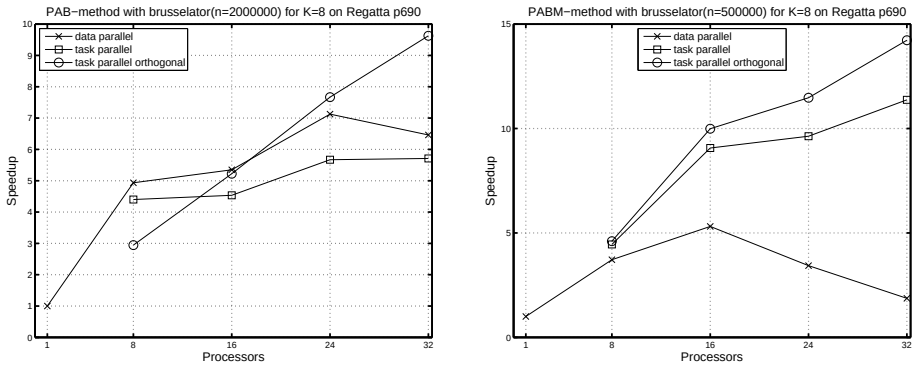

Figure 5: Speedups of the PAB (left) and PABM (right) methods for Brusselator on IBM Regatta with $K = 8$.

task parallelism. The runtimes for $p = 24$ are worse compared to the results for $p = 16$ because two processes need to be started on some nodes making the network interface on these nodes a bottleneck. For $p = 32$ the amount of data per node decreases leading to faster execution times. There is no speedup for the task parallel version because the communication overhead outweighs the additional computational power.

Figure 4 (right) shows the execution times of the PABM method on the CLiC cluster. This cluster is built from 528, 800 MHz, Pentium III processors connected by a fast-Ethernet network. For this cluster, the task parallel implementation is significantly faster than the data parallel implementation which is further improved by exploiting orthogonal communication structures. The impressive decrease in runtime when using concurrent multiprocessor tasks instead of data parallelism can be explained by the large communication overhead for collective communication operations on the CLiC due to its interconnection network. From the figure, it can be seen that for a larger number of processors, the task parallel implementations with orthogonal communication (as it is supported by the CM-task model) usually leads to the fastest runtimes.

Figure 5 shows the speedups of the different program versions for the PAB and PABM methods for an IBM Regatta system; this system uses 32, 1.7 GHz, Power4 processors per SMP node and has 41 nodes. The results show that the orthogonal program version can outperform a data parallel execution scheme even on shared memory platforms. The PABM method requires a higher computational effort compared to the PAB method and therefore also higher speedups are possible. Group based communication also plays a more important role in the PABM method, leading to a decrease of the speedups for the data parallel version for more than 16 processors.

The speedups for the PAB method on the CHiC cluster are presented in Figure 6 (left) for the sparse Brusselator system and in Figure 6 (right) for the dense Schrödinger system. The Schrödinger system uses a right-hand side function **f** for which the evaluation of each component depends on all components of its argument vector and therefore the evaluation time of the entire function **f** depends quadratically on the size of the ODE system. The CHiC cluster consists of 538 dual Opteron 2218 nodes clocked at 2.6 GHz interconnected by a 10GBit/s Infiniband network. For the benchmark tests the MVA-

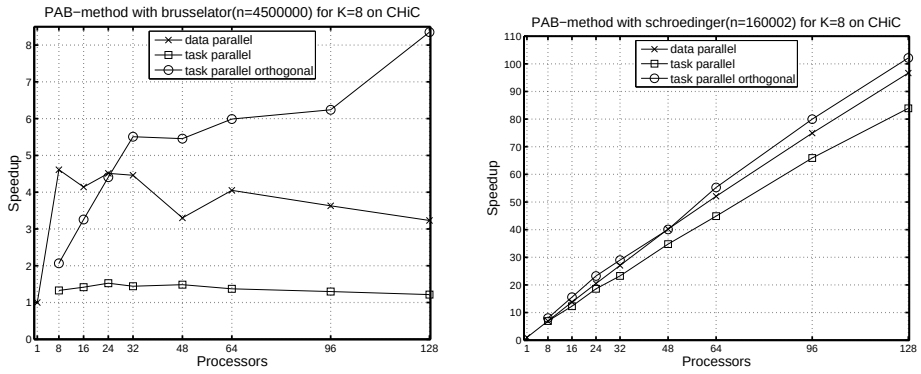

Figure 6: Speedups of the PAB-method with $K = 8$ on the CHiC with Infiniband network using a sparse ODE system (left) and a dense ODE system (right).

PICH2 MPI library was used. The computation to communication ratio of the dense system is much higher compared to a sparse system leading to much higher speedups. The number of executed arithmetic operations per node is identical in all three program versions and therefore the speedups for the dense system lie much closer together. For the sparse system, the achieved speedups are limited because the amount of communication and computation are of the same order of magnitude.

Altogether, the results show that the orthogonal program version, as one example for communication between CM-tasks, outperforms both other program version in almost all cases. Especially for cluster systems with a slower interconnection network, such as the CLiC cluster (see Figure 4 (right)) optimizations such as orthogonal task parallel versions are required to achieve competitive performance results. But also for platforms with a fast interconnection network like the CHiC cluster, significant performance improvements can be obtained, especially for a larger number of processors.

5 Related Work

In the past decade, several research groups have proposed models for mixed task and data parallel executions with the goal to obtain parallel programs with faster execution time and better scalability properties, see [2, 18] for an overview of systems and approaches and see [3] for a detailed investigation of the benefits of combining task and data parallel executions. An exploitation of task and data parallelism in the context of a parallelizing compiler with an integrated scheduler can be found in the Paradigm project [9, 14]. The approach in this article is an extension of these approaches which captures additional communication patterns.

Other environments for mixed parallelism in scientific computing are language extensions, see [6] for an overview. In contrast to our approach, these environments leave the task placement, i.e. the scheduling, to the programmer and do not have an explicit specification language. The Fx compiler[19] extends the HPF data parallel language with statements that allow the partitioning of processor groups into disjoint subgroups

whose size may be determined at runtime offering a semi-dynamic execution. [4] describes a concept to combine the task parallel Fortran M with the data parallel Fortan D or HPF to derive a mixed parallel execution. This concept allows communication between concurrently running parallel programming parts but lacks an automatic data re-distribution between data parallel tasks. Opus[5] uses Shared Data Abstractions (SDAs) for synchronization and communication between parallel program parts. The Tlib library [17] is a realization of the TwoL model as runtime system.

Scheduling algorithms for computing an appropriate mix of task and data parallel executions for M-task programs are presented in [21, 22]. For the decision, the scalability characteristics of the M-tasks and the communication costs between the M-tasks are taken into account. A comparison of different scheduling algorithms for M-task programs is given in [13]. These scheduling algorithms cannot be applied directly to CM-task programs, since they do not capture the C-relations between CM-tasks.

The use of skeletons to coordinate different program parts was considered within the Lithium environment [1]. Task and data parallel skeletons are available and can be nested within each other. Skeletons were also used in the COLT_{HPF}[12] compiler to create mixed parallel coordination programs providing a runtime system that controls communication and supports the dynamic loading of additional tasks. A lot of research has been invested in the development of the BSP (bulk synchronous parallelism) model and there exists a programming library (Oxford BSP library) that allows the formulation of BSP programs in an SPMD style [8]. NestStep extends the BSP model by supporting group-oriented parallelism by nesting of supersteps and a hierarchical processor group concept [10]. NestStep is defined as a set of extensions to existing programming languages like C or Java and is designed for a distributed address space.

6 Conclusions

In this paper, we have presented a parallel programming model with mixed task and data parallelism for coding modular applications. This model is based on M-tasks where each M-task is a parallel program part which can be executed on an arbitrary set of processors and can be hierarchically decomposed into further M-tasks. Programming models for M-tasks usually consider task graphs with control or data dependencies (precedence constraints). We have extended the M-task model by communication between concurrently running M-tasks. The model is able to capture communication between M-tasks, thus providing a flexible way to structure complex modular applications. In particular, the model is able to structure the communication between M-tasks such that orthogonal communication patterns can be exploited. Experimental results for solution methods for ODEs show a significant performance improvement compared to data parallel or pure task parallel execution schemes. Another area of examples which are expected to benefit from the CM-task model are modular simulation algorithms, e.g., from atmospheric simulation. For the implementation of efficient programs in the CM-task model, we have proposed a step-wise transformation process that is realized by a transformation framework. This framework supports the development of efficient CM-task programs by an automated transformation process and a toolset of interacting software tools to transform a specification into an executable program.

References

[1] M. Aldinucci, M. Danelutto, and P. Teti. An advanced environment supporting structured parallel programming in Java. *Future Generation Computer Systems*, 19(5):611–626, 2003.

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

[3] S. Chakrabarti, J. Demmel, and K. Yelick. Modeling the benefits of mixed data and task parallelism. In *Symposium on Parallel Algorithms and Architecture*, pages 74–83, 1995.

[4] M. Chandy, I. Foster, K. Kennedy, C. Koelbel, and C.-W. Tseng. Integrated support for task and data parallelism. *The Int. Journal of Supercomputer Applications*, 8(2):80–98, 1994.

[5] B. Chapman, M. Haines, P. Mehrota, H. Zima, and J. Van Rosendale. Opus: A coordination language for multidisciplinary applications. *Sci. Program.*, 6(4):345–362, 1997.

[6] S.J. Fink. *A Programming Model for Block-Structured Scientific Calculations on SMP Clusters*. PhD thesis, University of California, San Diego, 1998.

[7] E. Hairer, S.P. Nørsett, and G. Wanner. *Solving Ordinary Differential Equations I: Nonstiff Problems*. Springer–Verlag, Berlin, 1993.

[8] M. Hill, W. McColl, and D. Skillicorn. Questions and Answers about BSP. *Scientific Programming*, 6(3):249–274, 1997.

[9] P. Joisha and P. Banerjee. PARADIGM (version 2.0): A New HPF Compilation System. In *Proc. 1999 International Parallel Processing Symposium (IPPS'99)*, 1999.

[10] C.W. Keßler. NestStep: Nested Parallelism and Virtual Shared Memory for the BSP model. *The Journal of Supercomputing*, 17:245–262, 2001.

[11] M. Kühnemann, T. Rauber, and G. Rünger. Optimizing MPI Collective Communication by Orthogonal Structures. *Journal of Cluster Computing*, 9(3):257–279, 2006.

[12] S. Orlando, P. Palmerini, and R. Perego. Coordinating HPF programs to mix task and data parallelism. In *SAC '00: Proceedings of the 2000 ACM symposium on Applied computing*, pages 240–247. ACM Press, 2000.

[13] A. Radulescu, C. Nicolescu, A. van Gemund, and P.P.Jonker. CPR: Mixed task and data parallel scheduling for distributed systems. In *Proceedings of the 15th International Parallel and Distributed Symposium*, 2001.

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

[15] T. Rauber and G. Rünger. A Transformation Approach to Derive Efficient Parallel Implementations. *IEEE Transactions on Software Engineering*, 26(4):315–339, 2000.

[16] T. Rauber and G. Rünger. Execution Schemes for Parallel Adams Methods. In *Proc. of Euro-Par 2004*, pages 708–717. Springer LNCS 3149, 2004.

[17] T. Rauber and G. Rünger. Tlib - A Library to Support Programming with Hierarchical Multi-Processor Tasks. *J. of Parallel and Distributed Computing*, 65(3):347–360, 2005.

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

[19] J. Subhlok and B. Yang. A new model for integrated nested task and data parallel programming. In *Proceedings of the sixth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 1–12. ACM Press, 1997.

[20] P.J. van der Houwen and E. Messina. Parallel Adams Methods. *J. of Comp. and App. Mathematics*, 101:153–165, 1999.

[21] N. Vydyanathan, S. Krishnamoorthy, G. Sabin, U. Catalyurek, T. Kurc, P. Sadayappan, and J. Saltz. An integrated approach for processor allocation and scheduling of mixed-parallel applications. In *Proc. of the 2006 International Conference on Parallel Processing (ICPP'06)*. IEEE, 2006.

[22] N. Vydyanathan, S. Krishnamoorthy, G. Sabin, U. Catalyurek, T. Kurc, P. Sadayappan, and J. Saltz. Locality conscious processor allocation and scheduling for mixed parallel applications. In *Proc. of the 2006 IEEE Int. Conf. on Cluster Computing*. IEEE, 2006.