J. Parallel Distrib. Comput. 73 (2013) 220–234

Contents lists available at SciVerse ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

Programming support and scheduling for communicating parallel tasks

Jörg Dümmler [a,*], Thomas Rauber [b], Gudula Rünger [a]

[a] Chemnitz University of Technology, Department of Computer Science, 09111 Chemnitz, Germany
[b] Bayreuth University, Angewandte Informatik II, 95440 Bayreuth, Germany

ARTICLE INFO

Article history:
Received 13 March 2012
Received in revised form
6 September 2012
Accepted 27 September 2012
Available online 12 October 2012

Keywords:
Parallel tasks
Scheduling
Mixed parallelism
Algorithms
Scalability
Tool support

ABSTRACT

Task-based programming models are beneficial for the development of parallel programs for several reasons. They provide a decoupling of the specification of parallelism from the scheduling and mapping to execution resources of a specific hardware platform, thus allowing a flexible and individual mapping. For platforms with a distributed address space, the use of parallel tasks, instead of sequential tasks, adds the additional advantage of a structuring of the program into communication domains that can help to reduce the overall communication overhead.

In this article, we consider the parallel programming model of communicating parallel tasks (CM-tasks), which allows both task-internal communication as well as communication between concurrently executed tasks at arbitrary points of their execution. We propose a corresponding scheduling algorithm and describe how the scheduling is supported by a transformation tool. An experimental evaluation using synthetic task graphs as well as several complex application programs shows that employing the CM-task model may lead to significant performance improvements compared to other parallel execution schemes.

1. Introduction

Task-based approaches have the advantage to allow a decoupling of the computation specification for a given application algorithm from the actual mapping and execution on the computation resources of a parallel target platform. The programmer is then only responsible for the specification of the tasks of the application algorithm and their interactions. The mapping onto execution resources is usually based on a runtime prediction model and supported by a compiler tool or runtime system. Thus, the programmer is relieved from providing an explicit mapping. Moreover, the runtime system can select a suitable task mapping depending on the characteristics of the target platform, providing a portability of the application performance.

Many different variations of task-based programming systems have been investigated. An important distinction is whether the individual tasks are executed sequentially on a single execution resource (called single-processor tasks, *S*-tasks) or whether they can be executed on multiple execution resources (called parallel tasks, malleable tasks, or multi-processor tasks, *M*-tasks). *S*-tasks are often used for program development in shared address spaces, including single multi-core processors, and allow a flexible program development. Efficient load balancing methods can easily be integrated into the runtime system. Examples for such approaches are the task concepts in OpenMP 3.0 [25], Cilk [13], SMPSs [27], FG [6] for out of core algorithms, the TPL library for .NET [21], or the KOALA framework [16], which provides adaptive load balancing mechanisms. For distributed address spaces, the main challenge is to obtain a distributed load balancing of tasks with a low communication overhead. Thus, for load exchange between different address spaces, tasks should not be too fine-grained to avoid heavy communication traffic. An adaptive load balancing technique for this scenario based on work stealing has been presented in [7]. An example for an *S*-task runtime system for a distributed memory environment is ClusterSs [36].

Parallel tasks are typically more coarse-grained than *S*-tasks, since they are meant to be executed by an arbitrary number of execution resources. These execution resources may need to exchange data during the execution of a parallel task, and thus each parallel task may also comprise task-internal communication. In the standard parallel task model, the interactions between different parallel tasks are captured by input–output relations only, i.e., one parallel task may produce output data that is then used as an input for another parallel task. In this case, the two parallel tasks have to be executed one after another. Some of the parallel tasks of the program may also be independent of each other and provide the possibility for a concurrent execution on disjoint sets of execution resources leading to a mixed parallel execution. This parallel programming model is used, for example, by the Paradigm compiler [30], the TwoL model [32], and many other approaches [1,8,34].

* Corresponding author.
E-mail address: djo@cs.tu-chemnitz.de (J. Dümmler).

0743-7315/$ – see front matter © 2012 Elsevier Inc. All rights reserved.
doi:10.1016/j.jpdc.2012.09.017

In this article, we consider an extended parallel programming model called communicating M-tasks (CM-tasks) which additionally allows communication between parallel tasks that are executed concurrently. This additional kind of interaction between parallel tasks provides more flexibility for the structuring of a parallel application and provides the possibility for a more efficient organization of these data exchanges. This is especially beneficial for solvers for ordinary differential equations and multi-grid solvers. On the other hand, it also leads to additional restrictions of the execution order, since parallel tasks communicating with each other during their execution have to executed concurrently and cannot run one after another. As a consequence, new scheduling and load balancing methods are required for CM-task programs.

The contributions of this article include a detailed discussion of the CM-task programming model along with a comparison with standard parallel tasks and the proposal of a compiler framework that supports the development of CM-task applications. The framework generates an efficient executable MPI program from a CM-task specification provided by the application programmer. The core component of the framework is the static scheduler, which includes different scheduling algorithms that have been adapted to fit the requirements of the CM-task model. The article defines the underlying scheduling problem and describes these algorithms in detail. An experimental evaluation for several complex application programs shows that the scheduling algorithm for CM-tasks can lead to significant performance improvements compared to execution schemes resulting from other schedules. Applications from scientific computing offering a modular structure of different program components can benefit considerably from the CM-task model and the programming support.

The rest of the article is organized as follows. Section 2 describes the CM-task programming model. A compiler tool supporting the development of CM-task applications is presented in Section 3. The CM-task scheduling problem is defined in Sections 4 and 5 proposes an appropriate scheduling algorithm. Section 6 presents an experimental evaluation. Section 7 discusses related work and Section 8 concludes the article.

2. CM-task programming model

This section discusses the programming with CM-tasks and highlights the benefits of CM-tasks over standard parallel tasks.

2.1. Structure of CM-task programs

The CM-task programming model exhibits two well-separated 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 where each CM-task implements a specific part of the application in a way that an execution on an arbitrary number of execution resources is possible. Each CM-task operates on a set of input variables that it expects upon its activation and produces a set of output variables that are available after its termination. Additionally, there may be communication phases in which data is exchanged between two or more CM-tasks that are executed at the same time on disjoint sets of execution resources.

A CM-task can be a parallel module performing parallel computations (basic CM-task), e.g., a data parallel matrix multiplication, or can have an internal structure activating other CM-tasks (composed CM-task). The internal parallelism of basic CM-tasks is realized using 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. In the following, we assume that each CM-task is executed by a number of MPI processes, i.e., task-internal data exchanges are implemented with MPI. This implies that each CM-task defines a data distribution among the executing processes for each structured input or output variable used.

Dependences between CM-tasks resulting from input/output variables and the communication phases defined are captured by the following relations:

- *P-relation*: A P-relation (precedence relation) from a CM-task A to a CM-task B exists if A provides output data required by B as input before B can start its execution. This relation is not symmetric and is denoted by $A\delta_P B$.
- *C-relation*: A C-relation (communication relation) between CM-tasks A and B exists, if A and B have to exchange data during their execution. This relation is symmetric and is denoted by $A\delta_C B$.

P-relations capture input–output dependences between tasks and require the respective CM-tasks to be executed one after another. Moreover, a data re-distribution operation may be required between CM-tasks A and B with $A\delta_P B$. This is the case if A and B are executed on different sets of processors or if A produces its output data in a different data distribution then it is expected by B. C-relations capture communication phases in which intermediate results are exchanged and enforce a concurrent execution of the respective CM-tasks. The communication operations to realize these data transfers are included in the respective CM-tasks, such that optimized communication patterns can be exploited. The runtime system has to provide a common communication context for all CM-tasks participating in the same communication phase, e.g., by providing a suitable MPI communicator.

A CM-task program can be described by a CM-task graph $G = (V, E)$ where the set of nodes $V = \{A_1, \ldots, A_n\}$ represents the set of CM-tasks and the set of edges E represents the (C and P) relations between the CM-tasks. The set E can be partitioned into two disjoint sets E_C and E_P with $E = E_P \cup E_C$. E_P contains directed edges representing the P-relations defined between CM-tasks. There is a precedence edge from CM-task A to CM-task B in E_P if an input–output relation from A to B exists. E_C contains bidirectional edges representing the C-relations defined between CM-tasks. An example for a CM-task graph is shown in Fig. 1(a).

2.2. Comparison of CM-tasks with standard parallel tasks

The CM-task programming model supports two kinds of interactions between CM-tasks: P-relations between data dependent tasks and C-relations between tasks that communicate with each other during their execution. In contrast, programming models based on standard parallel tasks such as Paradigm [30] and TwoL [32] only support P-relations between the tasks. In the following, we show that the additional C-relations in the CM-task model allow a more flexible formulation of the tasks compared to the standard parallel task model. As a case study, we consider time stepping methods as they are often used for the numerical solution of systems of ordinary differential equations (ODEs).

A typical task graph for these methods using only P-relations is shown in Fig. 1(b). This standard parallel task graph shows two time steps where parallel tasks M_2, M_3, and M_4 perform independent computations for the first time step, and parallel tasks M_6, M_7, and M_8 perform analogous computations for the next time step. In between, M_5 combines the results, e.g., for error control or information exchange. In the standard parallel task model, M_2 and M_6 cannot be combined because the result of M_2 is used by M_5. As a consequence, the parallel tasks used may be too fine-grained possibly resulting in a significant management overhead.

In the CM-task model, however, a single CM-task can implement the computations of multiple time steps, see Fig. 1(c). For example, CM-task CM_2 performs the computations of parallel tasks

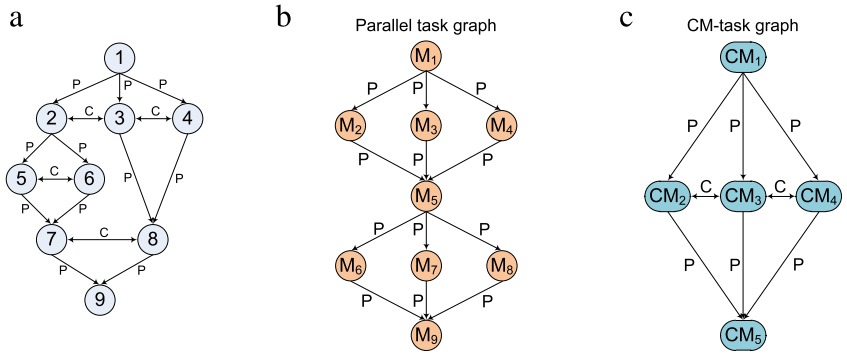

Fig. 1. (a) Example for a CM-task graph with precedence edges (annotation p) and communication edges (annotation c). (b) Dependence structure between the tasks of a typical ODE solver in the form of a standard parallel task graph using P-relations only. (c) Possible CM-task graph for a typical ODE solver using both P-relations and C-relations.

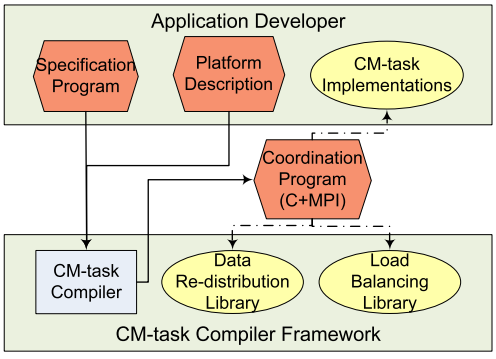

Fig. 2. Overview of the CM-task compiler framework. The user provides specifications for the parallel application and the parallel platform and implements the basic CM-tasks. The CM-task compiler translates the user-provided specifications into an executable coordination program that contains an efficient implementation of the application on the target platform specified.

M_2 and M_6, and also computes the results at the end of the time step along with CM-tasks CM_3 and CM_4. The communication operations required to exchange the intermediate results at the end of the time step are captured by appropriate C-relations. For the implementation of these data transfers, orthogonal communication patterns [31] can be exploited, which leads to a reduction of the communication overhead, see also the benchmark results presented in Section 6.

3. Programming support for CM-task applications

A major advantage of the CM-task programming model is its flexibility to adapt the execution of an application to the characteristics of the target platform, e.g., by selecting an appropriate execution order for independent CM-tasks. But such an adaption may be complex and error-prone especially for large application programs. Therefore, the CM-task compiler framework has been designed to assist the application developer by providing scheduling and load balancing methods as well as a generator for a platform-specific implementation of a CM-task program.

Fig. 2 shows an overview of the framework. As input, the application developer has to provide

- a platform-independent specification program that describes the high-level task structure of the parallel application, see an example in Fig. 4,
- a platform description that defines a homogeneous target machine by specifying a number of hardware parameters such as the number of processors, the speed of the processors, and the speed of the interconnection network, and
- a set of basic CM-tasks that are provided as parallel functions to be executed on an arbitrary number of processors, e.g., using C + MPI or a hybrid C + MPI + OpenMP model.

The CM-task compiler translates the specification program and the platform description into an executable C + MPI coordination program, see Section 3.3 for a detailed description of

$M \rightarrow$ **seq** $\{ M_1 M_2 \ldots M_n \}$	/* consecutive execution */
| **par** $\{ M_1 M_2 \ldots M_n \}$	/* independent computations */
| **for** $(i = 1 : n) \{ M_1 \}$	/* loop with data dependencies */
| **while** $(cond)\#It \{ M_1 \}$	/* loop with data dependencies */
| **parfor** $(i = 1 : n) \{ M_1 \}$	/* loop with independent iterations */
| **if** $(cond) \{ M_1 \}$	/* conditional execution */
| **if** $(cond) \{ M_1 \}$ **else** $\{ M_2 \}$	/* conditional execution */
| C	
$C \rightarrow$ BC $(a_1, \ldots, a_n)$;	/* execution of a basic CM-task */
| CC $(a_1, \ldots, a_n)$;	/* execution of a composed CM-task */
| **cpar** $\{ C_1 C_2 \ldots C_n \}$	/* concurrent execution */
| **cparfor** $(i = 1 : n) \{ C_1 \}$	/* concurrent execution of iterations */

Fig. 3. Grammar for the specification of the available task parallelism within a composed CM-task (simplified).

```
1  const K=8;                              // number of stage vectors
2  const n=...;                            // ODE system size
3
4  // data type and data distribution type definitions [...]
5
6  // basic CM-task definitions
7  cmtask initstep (x,h:scalar:out) runtime [...];
8  cmtask pabmstep (k:int, x,h:scalar:in, y_k:vector:inout:block,
9    y_k1:vector:in:block, ort:vector:comm) runtime [...];
10 cmtask updatestep (x,h:scalar:inout) runtime [...];
11
12 // composed CM-task definitions
13 cmmain pabm (X:double,y:vecs:inout:replic) {
14  // declaration of local variables
15  var x : scalar;                 // current time index
16  var h : scalar;                 // current step size
17  var ortcomm : vecs;             // intermediate results
18  // module expression
19  seq {
20   initstep(x,h);
21   while (x[0] < X)#100 {
22    seq {
23     cparfor (k = 0:K-1) {
24       pabmstep (k,x,h,y[k],y[K-1],ortcomm); }
25     updatestep (x,h);
26 }}}}
```

Fig. 4. Specification program for the PABM method.

the transformation process. At runtime, the coordination program is responsible for (i) the actual creation of the required processor groups, (ii) the data re-distribution operations to guarantee the correct distribution of input data before starting a CM-task, and (iii) the actual execution of the user-provided parallel functions implementing the basic CM-tasks on the processor groups as defined by the computed schedule. The data re-distribution operations are performed by a separate library that is provided as part of the CM-task compiler framework. Two different approaches are supported for the generation of the coordination program.

- The *static approach* of the CM-task compiler uses a fixed schedule, i.e., both, the execution order and the executing processor groups of the CM-tasks are fixed at compile time and cannot be changed at runtime. This approach is especially suited for dedicated homogeneous platforms and requires an accurate cost model for a good schedule. The fixed schedule enables several static optimizations, such as the precomputation of the communication pattern for data re-distribution operations at compile time.
- The *semi-dynamic approach* of the CM-task compiler combines a static schedule with dynamic load balancing. The static schedule defines the execution order of the CM-tasks as well as the initial processor groups used to execute the CM-tasks. The semi-dynamic coordination program produced includes profiling code that measures the execution times of the CM-tasks at runtime of the application. The dynamic load balancing library of the CM-task compiler framework adapts the sizes of the processor groups based on the runtimes measured. The semi-dynamic approach is especially suited for non-dedicated heterogeneous platforms.

The compiler approach employed is fully transparent for the user-supplied specification program and CM-task implementations, i.e., the application developer does not need to modify the implementation when switching from the static to the semi-dynamic approach or vice versa. In this article, we focus on the static approach; the semi-dynamic approach is described in [9].

3.1. Specification language

The platform-independent specification program defines the available basic CM-tasks along with a cost estimation, the internal structure of the composed CM-tasks as well as the data types and data distribution types used for the input and output parameters of the CM-tasks. Supported data types encompass scalars and multi-dimensional array structures. A data distribution type can either be an arbitrary block-cyclic distribution over a multi-dimensional processor mesh or a replicated storage on an arbitrary subset of the processors.

The definition of a basic CM-task starts with the keyword *cmtask* followed by a unique name, a parameter list and a cost expression. The parameter list includes input and output parameters with their respective data types and data distribution types as well as special parameters that are communicated along the C-relations. The cost expression is defined as a symbolic formula in closed form depending on the number of executing processors p and platform specific parameters whose values are provided in the separate machine description input, see Section 3.2 for a more detailed discussion of the cost model.

Composed CM-tasks are defined by using the keyword *cmgraph* followed by a name, a parameter list similar to basic CM-tasks, and a hierarchical module expression. One distinguished

composed CM-task represents the entire application; this CM-task is denoted by using the keyword *cmmain* instead of *cmgraph*. The module expression consists of activations of (basic or composed) CM-tasks and predefined operators which define the maximum degree of task parallelism that may be exploited by the CM-task compiler framework. Operators are available to define a consecutive execution, to define a concurrent execution, and to define data independence such that the framework can select a suitable execution order. Fig. 3 gives an overview of the available operators. The *P*-relations and *C*-relations of the CM-task program are defined implicitly using appropriate variable names in the parameter lists of the CM-task activations.

Example specification. As an example application we consider the implicit parallel Adams–Bashforth–Moulton (PABM) method [37] which is suitable for solving stiff systems of ordinary differential equations (ODEs) [15]. The PABM method performs a large number of time steps that have to be executed one after another due to data dependences between successive steps. Within a time step, K stage vectors are computed and then combined to the final approximation of this step.

Fig. 4 shows an appropriate CM-task specification program. The data type and data distribution type definitions have been omitted to improve readability. Three basic CM-tasks are defined in lines 7–10: the CM-task `initstep` initializes the first time step, the CM-task `pabmstep` computes one of the K stage vectors for a single time step, and the CM-task `updatestep` updates the step size for the next time step. The CM-task `pabmstep` has three input parameters (`x` and `h` with data type `scalar`, and `y_k1` with data type `vector` and data distribution type `block`), one input/output parameter (`y_k`), and one parameter that is used to exchange data during its execution with other CM-tasks (`ort`). The cost expressions associated with the basic CM-tasks are omitted and will be discussed in Section 3.2.

The module expression of the composed CM-task `pabm` defines an activation of CM-task `initstep` (line 20) and a *while*-loop (lines 21–25) that have to be executed one after another due to the *seq*-operator on line 19. The loop is executed until the current time index `x` reaches a predefined limit `X`. The term `#100` on line 21 defines an estimate of the number of iterations and is used by the CM-task compiler framework to predict the resulting execution time of the entire application. The body of the *while*-loop contains a *cparfor*-loop which defines K activations of CM-task `pabmstep` that have to be executed concurrently. All K activations are connected by *C*-relations, since the communication parameter `ortcomm` is passed to each one. The *seq*-operator on line 22 defines that CM-task `updatestep` cannot be executed until the entire *cparfor*-loop has been terminated.

3.2. Cost model

In the specification program, the costs for basic CM-tasks are described in form of *symbolic runtime formulas* [3,20]. A symbolic runtime formula T_A for a basic CM-task A is a function whose structure reflects the computation and communication operations performed by A including the data exchanges along the *C*-relations. T_A typically has the form

$$T_A(p) = \frac{ops(A)}{p} * T_{op} + T_{comm}(A, p),$$

where $ops(A)$ is the number of arithmetic operations of A, T_{op} is the average execution time of an arithmetic operation on the execution platform, and $T_{comm}(A, p)$ is the sum of the internal communication times of A when executed on p processors.

The communication time $T_{comm}(A, p)$ depends on the number and type of the communication operations inside A and is specified using cost formulas for the communication primitives provided by the MPI library. For example, the execution time of a broadcast operation that uses a binomial tree of depth $\log(p)$ can be estimated by a function

$$T_{bc}(p, b) = (\tau + t_c * \log(p)) * b,$$

where p is the number of processors participating in the operation, b is the amount of data to be transferred, and τ and t_c are hardware-specific parameters. These formulas are provided in the separate platform description input file.

The symbolic runtime formula for a specific basic CM-task A can either be obtained by hand, e.g., by fitting measured execution times to a function prototype, or automatically extracted from the source code of A by a suitable compiler tool [19]. For example, the execution time of the basic CM-task `pabmstep` that computes a single stage vector in a K-stage PABM method, see Fig. 4, can be described by

$$\begin{aligned} T_{\texttt{pabmstep}}(p) = {} & (I+1) * \frac{d}{p} * T_{eval} + (2K+1+3I) * \frac{d}{p} * T_{op} \\ & + T_{ag}\left(K, \frac{d}{p}\right) + T_{bc}\left(K, \frac{d}{p}\right) \\ & + (I+1) * T_{ag}\left(\frac{p}{K}, \frac{d}{p}\right). \end{aligned}$$

In this formula, I denotes the number of fixed point iterations performed by the PABM method, T_{eval} defines the time required to evaluate a single ODE of the d-dimensional ODE system, and $T_{ag}(p, b)$ is a cost prediction for the execution of a multi-broadcast (`MPI_Allgather()`) operation depending on the number of participating processes p and the amount of data b to be transmitted.

The costs for composed CM-tasks are built up from the costs of the basic CM-tasks and the communication times for the *P*-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 the cost formulas of CM_1 and CM_2 and the data re-distribution costs between these CM-tasks is used. The costs for the CM-task *cmmain* determine the costs for the entire application.

3.3. CM-task compiler

The CM-task compiler performs several transformation steps to translate a specification program into an executable C + MPI coordination program that is adapted to a specific parallel platform, see Fig. 5 for an overview. The transformation steps include the detection of data dependences, the computation of a platform-dependent static schedule, the insertion of data re-distribution operations, and the translation into the coordination program. In the following, we describe the phases in detail.

The *dataflow analyzer* detects the *P*-relations and the *C*-relations between the activations of the CM-tasks. A *P*-relation is inserted between CM-tasks A and B that have to be executed one after another due to a *seq*-operator in case A has an output parameter that is used as an input for B. A *C*-relation is inserted between CM-tasks A and B that have to be executed concurrently due to a *cpar*- or *cparfor*-operator if A and B have a common communication parameter.

The *static scheduler* performs the scheduling in the following three steps.

(1) The specification program is transformed into a set of CM-task graphs. A CM-task graph is constructed for each body of a *for*- or *while*-loop, each branch of the *if*-operator, and for each composed CM-task graph. The *parfor*- and *cparfor*-loops are unrolled such that different scheduling decisions can be made

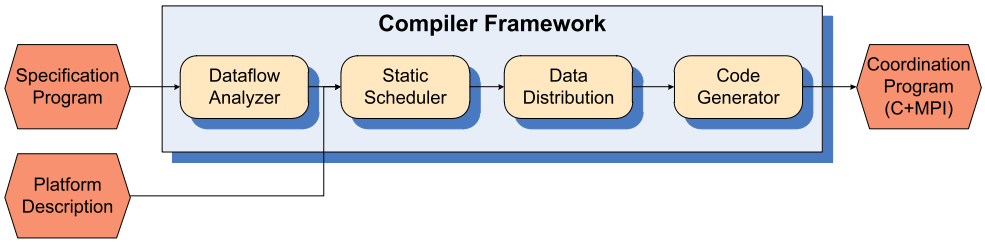

Fig. 5. Transformation steps performed by the CM-task compiler to generate a C + MPI coordination program from a user-provided specification program and platform description.

for each iteration of these loops. Note that the bounds of these loops need to be known at compile time. The resulting CM-task graphs are organized hierarchically according to the nesting of the corresponding operators. For example, the specification program from Fig. 4 is translated into two CM-task graphs: an upper-level CM-task graph with two nodes representing the function `initstep` and the entire *while*-loop, respectively, and a lower-level CM-task graph for the body of the *while*-loop that contains $K + 1$ nodes where K nodes represent the instances of function `pabmstep` and one node represents `updatestep`.

(2) Next, a feasible CM-task schedule (as defined in Section 4.2) is produced for each CM-task graph. The scheduling starts with the CM-task graph representing the entire application and then traverses the hierarchy of CM-task graphs. The scheduling decisions on the upper levels determine the number of processors that are available for the lower levels. For example, the number of processors assigned to the entire *while*-loop is equal to the number of available processors for scheduling the loop body. The scheduling for a single CM-task graph is described in Section 5.

(3) The resulting CM-task schedules are transformed back into an extended specification program with additional annotations that define the processor groups for each CM-task activation. Additionally, the operators of the initial program are adjusted to reflect the actual execution order defined by the computed schedule, i.e., a *par* operator may be transformed into a *seq* operator if the schedule defines a consecutive execution for independent program parts.

The *data distribution* phase proceeds in two steps. First, it determines suitable data distribution types in which the variables are provided when entering a loop (*for* and *while*) or a conditional (*if*). For this purpose, a heuristic is used that tries to minimize the number of required data re-distribution operations inside the respective loop or conditional. Second, it inserts appropriate data re-distribution operations, such that the input data of each activated CM-task is provided in the correct data distribution. The final *code generation* phase uses the information provided by the previous phases to create the final output program. This phase is implemented by a top-down traversal of the abstract syntax tree using a syntax-directed translation scheme.

4. Scheduling of CM-task programs

This section defines the scheduling problem for CM-task programs and discusses the constraints resulting from the P-relations and C-relations between the CM-tasks of a program.

4.1. Cost annotations for CM-task graphs

For the definition of the scheduling problem, we assume that the CM-task graph $G = (V, E)$ of a CM-task program is annotated with cost information. The execution time of each CM-task is described by a function

$$T : V \times \{1, \ldots, q\} \to \mathbb{R}$$

where q is the size of the processor set Q of a (homogeneous) target platform. The runtime $T(A, |R|)$ of a CM-task A executed on a subset $R \subseteq Q$ comprises the computation time of A, the internal communication time, as well as the time for data exchanges with simultaneously running CM-tasks with which A has a C-relation.

The P-relation edges of the CM-task graph are associated with communication costs

$$T_P : E_P \times \{1, \ldots, q\} \times \{1, \ldots, q\} \to \mathbb{R}$$

where $T_P(e, |R_1|, |R_2|)$ with $e = (A_1, A_2)$ denotes the communication costs between CM-task A_1 executed on processor set R_1 and CM-task A_2 executed on processor set R_2 with $R_1, R_2 \subseteq Q$. These communication costs may result from a re-distribution operation that is required between CM-tasks A_1 and A_2 with $A_1 \delta_P A_2$ if $R_1 \neq R_2$ or if $R_1 = R_2$ and A_1 provides its output in a different data distribution as expected by A_2.

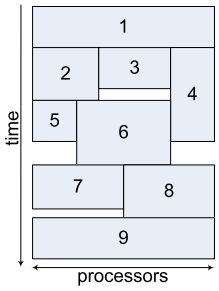

Fig. 6. Illustration of a possible CM-task schedule for the CM-task graph from Fig. 1(a).

4.2. Scheduling constraints

A schedule S of a given CM-task program maps each CM-task A_i, $i = 1, \ldots, n$, to an execution time interval with start time s_i and a processor set R_i with $R_i \subseteq Q$, i.e.,

$$S : \{A_1, \ldots, A_n\} \to \mathbb{R} \times 2^Q \quad \text{with } S(A_i) = (s_i, R_i).$$

An illustration of a CM-task schedule is given in Fig. 6. The P-relations and C-relations between the CM-tasks of a program lead to the following scheduling constraints:

(I) *Consecutive time intervals.* If there is a P-relation $A_i \delta_P A_j$ between two CM-tasks A_i and A_j, $i, j \in \{1, \ldots, n\}, i \neq j$, then the execution of A_j cannot be started before the execution of A_i and all required data re-distribution operations between A_i and A_j have been terminated. Thus, for the starting times s_i and s_j of A_i and A_j and executing sets R_i and R_j of processors, respectively, the following condition must be fulfilled:

$$s_i + T(A_i, |R_i|) + T_P\left(e, |R_i|, |R_j|\right) \leq s_j \quad \text{with } e = (A_i, A_j).$$

(II) *Simultaneous time intervals.* If there is a C-relation $A_i \delta_C A_j$ between A_i and A_j, then A_i and A_j have to be executed concurrently

(with overlapping execution time intervals) on disjoint sets of processors R_i and R_j, i.e., the following conditions must be fulfilled:

$$R_i \cap R_j = \emptyset \quad \text{and}$$

$$[s_i, s_i + T(A_i, |R_i|)] \cap \left[s_j, s_j + T(A_j, |R_j|)\right] \neq \emptyset.$$

The overlapping execution time intervals guarantee that the CM-tasks A_i and A_j can exchange data during their execution.

(III) *Arbitrary execution order.* If there are no P- or C-relations between CM-tasks A_i and A_j, then A_i and A_j can be executed in concurrent or in consecutive execution order. For a concurrent execution, disjoint processor sets R_i and R_j have to be used, i.e.,

$$\text{if } [s_i, s_i + T_g(A_i, |R_i|)] \cap [s_j, s_j + T_g(A_j, |R_j|)] \neq \emptyset$$
$$\text{then } R_i \cap R_j = \emptyset$$

where T_g comprises the execution time of A_i as well as the communication time for data exchanges with successively executed CM-tasks A_k with $A_i \delta_P A_k$, i.e.,

$$T_g(A_i, |R_i|) = T(A_i, |R_i|) + \sum_{k=1}^{n} T_P(e, |R_i|, |R_k|)$$
$$\text{with } e = (A_i, A_k) \in E_P.$$

In the following, a schedule that meets the constraints (I)–(III) is called *feasible*. A feasible schedule S leads to a total execution time $T_{\max}(S)$ that is defined as the point in time when all CM-tasks have been finished, i.e.:

$$T_{\max}(S) = \max_{i=1,\ldots,n} \{s_i + T_g(A_i, |R_i|)\}.$$

The problem of finding a feasible schedule S that minimizes $T_{\max}(S)$ is called scheduling problem for a CM-task program.

5. Scheduling algorithm

This section proposes a scheduling algorithm for CM-task graphs with annotated cost information that proceeds in four phases. In the first phase, CM-tasks that are connected by C-relations are combined to form a so-called *super-task*. The tasks within a super-task have to be executed concurrently due to communication between them, see Constraint (II) from Section 4.2. The aggregation of CM-tasks to super-tasks transforms the CM-task graph into a super-task graph with generalized P-relations between the super-tasks. The second phase determines a group layout within each super-task by using a load balancing algorithm. The third phase computes an execution order and determines processor groups for the super-tasks by taking the generalized P-relations of the super-task graph into account. This phase is similar to the scheduling of standard parallel tasks and, thus, a layer-based or a critical-path based approach can be utilized. The final phase combines the results of the previous steps and generates the final CM-task schedule. In the following, the phases of the algorithm are described in detail.

5.1. Transformation of the CM-task graph

The first phase identifies CM-tasks that are connected by C-relations and combines them into larger super-tasks as defined in the following.

Definition 1 (*Super-Task*)**.** Let $G = (V, E)$ be a CM-task graph. A super-task is a maximum subgraph $\hat{G} = (\hat{V}, \hat{E})$ of G with $\hat{V} \subseteq V$ and $\hat{E} \subseteq E_C$ such that each pair of CM-tasks $A, B \in \hat{V}$ is connected by a path of bidirectional edges in $\hat{E}$.

Each CM-task and each bidirectional C-relation edge of a CM-task graph belongs to exactly one super-task. A single CM-task without C-relations to any other CM-task forms a super-task by itself. The problem of finding the super-tasks of a CM-task graph is equivalent to discovering the connected components of an undirected graph, considering the C-relations as undirected edges. Using the super-tasks constructed, the CM-task graph is transformed into a super-task graph as defined next.

Algorithm 1: Load balancing for a single super-task.

```
1 begin
2     let Ĝ = (V̂, Ê) be a super-task with V̂ = {A_1, ..., A_m};
3     set L_Ĝ(A_i, m) = 1 for i = 1, ..., m;
4     for (p = m + 1, ..., q) do
5         set L_Ĝ(A_i, p) = L_Ĝ(A_i, p − 1) for i = 1, ..., m;
6         find CM-task A_k ∈ V̂ with maximum value of
              T(A_k, L_Ĝ(A_k, p − 1));
7         increase L_Ĝ(A_k, p) by 1;
```

Definition 2 (*Super-Task Graph*)**.** Let $G = (V, E)$ be a CM-task graph comprising l super-tasks $\hat{G}_1 = (\hat{V}_1, \hat{E}_1), \ldots, \hat{G}_l = (\hat{V}_l, \hat{E}_l)$. The super-task graph corresponding to G is a directed graph $G' = (V', E')$ with a set of l nodes $V' = \{\hat{G}_1, \ldots, \hat{G}_l\}$ and a set of directed edges $E' = \{(\hat{G}_i, \hat{G}_j) \mid \text{there exists } A \in \hat{V}_i, B \in \hat{V}_j \text{ with } A \delta_P B\}$.

Fig. 7(a) and (b) show an example CM-task graph with the corresponding super-task graph. Fig. 7(c)–(e) illustrates the scheduling phases described in the following subsections.

5.2. Load balancing for super-tasks

The second phase is an iterative load balancing algorithm shown in Algorithm 1 that determines the number of processors used to execute the CM-tasks inside a specific super-task $\hat{G}$. The algorithm tries to find an assignment such that the execution time of the entire super-task $\hat{G}$ is at a minimum. The load balancing decision depends on the number of processors available for the execution of $\hat{G}$. The super-task $\hat{G}$ can be executed on any number of processors between m and q where m is the number of CM-tasks inside $\hat{G}$ and q is the total number of processors available. The number of processors must be at least m because all m CM-tasks of a super-task have to be executed concurrently to each other. Since the exact number of processors available for super-task $\hat{G}$ is determined not until the next phase of the scheduling algorithm, all possible numbers p of processors are considered, i.e., $m \leq p \leq q$. The result of the load balancing algorithm is a super-task allocation $L_{\hat{G}}$ for $\hat{G}$, where $L_{\hat{G}}(A, p)$ specifies how many processors are allocated to CM-task A inside super-task $\hat{G}$ when p processors are available for the entire super-task $\hat{G}$.

Algorithm 1 starts with $p = m$ and assigns a single processor to each CM-task of the super-task $\hat{G}$ (line 3). In each step, the number of available processors p is increased by one and the additional processor is assigned to the CM-task A_k that has the largest parallel execution time within the current super-task allocation. This usually decreases the execution time of A_k, and another CM-task may then have the largest execution time.

An alternative method to determine the number of processors for each CM-task is to use the sequential execution time and to assign

$$L_{\hat{G}}(A_i, p) = p \cdot \frac{T(A_i, 1)}{\sum_{A_j \in \hat{V}} T(A_j, 1)}$$

processors to CM-task $A_i \in \hat{V}$. Compared to this approach, the iterative assignment of Algorithm 1 has the advantage that it takes scalability effects into account, captured by using the parallel execution time.

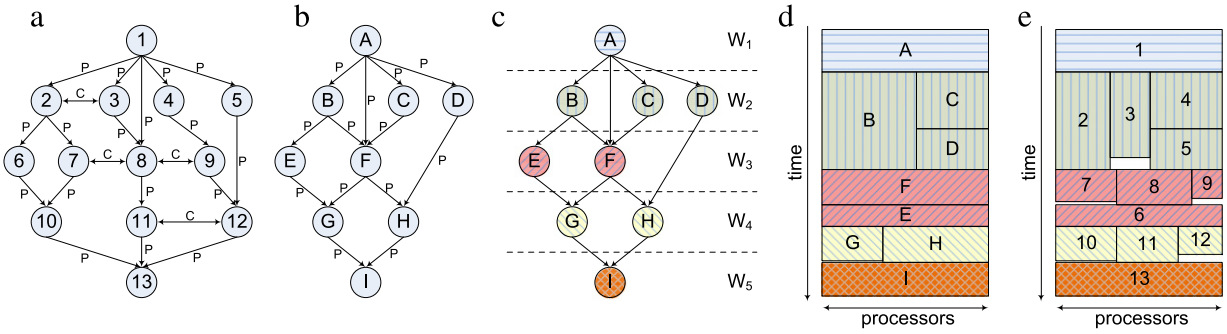

Fig. 7. Illustration of the scheduling algorithm for CM-task graphs. (a) Initial CM-task graph consisting of CM-tasks $\{1, \ldots, 13\}$. (b) Corresponding super-task graph with super-tasks $A = \{1\}$, $B = \{2, 3\}$, $C = \{4\}$, $D = \{5\}$, $E = \{6\}$, $F = \{7, 8, 9\}$, $G = \{10\}$, $H = \{11, 12\}$, and $I = \{13\}$. (c) Subdivision of the super-task graph into five consecutive layers $W_1 = \{A\}$, $W_2 = \{B, C, D\}$, $W_3 = \{E, F\}$, $W_4 = \{G, H\}$, and $W_5 = \{I\}$. (d) Possible schedule for the super-task graph consisting of a subschedule for each layer of the graph according to Algorithm 2. (e) Schedule for the original CM-task graph including load balancing from Algorithm 1.

5.3. Costs for super-task graphs

Using the super-task allocation $L_{\hat{G}}$, the cost of a super-task $\hat{G}$ can be calculated by the cost function defined below.

Definition 3 (*Costs For Super-Task Graphs*)**.** Let $G = (V, E)$ be a CM-task graph and $G' = (V', E')$ its corresponding super-task graph. A node $\hat{G} = (\hat{V}, \hat{E})$ of G' executed on p processors has costs

$$T'(\hat{G}, p) = \begin{cases} \infty & \text{if } p < |\hat{V}| \\ \max\limits_{A \in \hat{V}} T(A, L_{\hat{G}}(A, p)) & \text{otherwise.} \end{cases}$$

A directed edge $\hat{e}_{ij} = (\hat{G}_i, \hat{G}_j)$ with $\hat{G}_i = (\hat{V}_i, \hat{E}_i)$, $\hat{G}_j = (\hat{V}_j, \hat{E}_j)$, $i \neq j$, has costs

$$T'_P(\hat{e}_{ij}, p_i, p_j) = \sum_{e \in RE} T_P(e, L_{\hat{G}_i}(A, p_i), L_{\hat{G}_j}(B, p_j))$$

with $RE = \{e = (A, B) |$ there exists $A \in \hat{V}_i$, $B \in \hat{V}_j$ with $A \delta_P B\}$.

The cost information is needed for the scheduling algorithm for super-task graphs presented next.

5.4. Scheduling of the super-task graph

A super-task graph resembles a standard parallel task graph. However, there is an important difference: the scheduling problem for a super-task graph has the additional restriction that the number of processors assigned to a super-task must not be below the number of CM-tasks included in this super-task. This constraint guarantees that a concurrent execution of all CM-tasks within one super-task is possible. As a consequence, scheduling algorithms for parallel tasks have to be modified to take this requirement into account. The scheduling of parallel tasks is often performed by either a layer-based or a critical-path-based approach. In the following, we consider these two categories of algorithms in detail.

5.4.1. Layer-based scheduling of the super-task graph

Layer-based scheduling algorithms are well suited for parallel applications consisting of multiple consecutive phases each of which performing computations that can be captured by independent tasks. In this subsection, we propose a scheduling algorithm for a super-task graph G' that is based on a layer-based scheduling algorithm for parallel tasks [32] and additionally exploits the load balancing information from Algorithm 1. The new scheduling algorithm is called *CM-Layer* and proceeds in two steps.

In the first step, the super-task graph G' is partitioned into layers of independent super-task nodes such that the consecutive execution of the layers leads to a feasible schedule for the entire super-task graph. The partitioning is performed by a greedy algorithm that runs over the super-task graph in a breadth-first manner and puts as many super-tasks as possible into the current layer. An illustration is given in Fig. 7(c).

Algorithm 2: Scheduling algorithm for a single layer of the super-task graph.

```
1  begin
2      let W = {Ĝ_1, ..., Ĝ_r} be one layer of the super-task
           graph G' = (V', E') consisting of r CM-tasks;
3      let f = max_{i=1,...,r} |V̂_i| be the maximum number of CM-tasks
           in any super-task of W;
4      set T_min = ∞;
5      for (κ = 1, ..., min{q − f + 1, r}) do
6          partition the set Q of q = |Q| processors into disjoint
               subsets R_1, ..., R_κ such that
               |R_1| = max{⌈p/κ⌉, f} and R_2, ..., R_κ
7              have about equal size;
8          sort {Ĝ_1, ..., Ĝ_r} such that T(Ĝ_i, |R_1|) ≥ T(Ĝ_{i+1}, |R_1|)
           for i = 1, ..., r − 1;
9          for (j = 1, ..., r) do
10             assign Ĝ_j to the group R_l with the smallest
               accumulated execution time and |R_l| ≥ |V̂_j|;
11         adjust the sizes of the subsets R_1, ..., R_κ to reduce
               load imbalances;
12         T_κ = max_{j=1,...,κ} accumulated execution time of R_j;
13         if (T_κ < T_min) then T_min = T_κ;
```

In the second step, the layers are treated one after another and the scheduling algorithm given in Algorithm 2 is applied to each layer. The goal of the scheduling algorithm is to select a partition of the processor set into κ processor groups. Each of these groups is responsible for the execution of a specific set of super-tasks that are also selected by this algorithm. An illustration of such a group partitioning and a corresponding assignment of super-tasks is given in Fig. 7(d).

The scheduling algorithm for a single layer W with $r = |W|$ super-tasks tests all suitable values for the number κ of processor groups with $\kappa \leq r$ and selects the number of groups that leads to the smallest overall execution time (line 5). For a specific value of κ, the set of q processors is partitioned into subgroups such that at least one of the groups is large enough to execute any super-task of W. In particular, the largest processor group R_1 contains at least f processors where f denotes the maximum number of tasks which any of the super-tasks contains in its node set $\hat{V}_1, \ldots, \hat{V}_r$ (line 3). If $f \leq q/\kappa$ then a distribution into κ processor groups of equal size is chosen (line 6). If $f > q/\kappa$ then one processor group is made

large enough to contain exactly f processors and the rest of the processors is evenly partitioned into $\kappa - 1$ processor groups. For the assignment of super-tasks to processor groups, a list scheduling algorithm is employed that considers the super-tasks one after another in decreasing order of their estimated execution time (line 7). The group R_l of processors for a specific super-task $\hat{G}_j$ is selected such that R_l is large enough to execute $\hat{G}_j$ and assigning $\hat{G}_j$ to R_l leads to the overall smallest accumulated execution time (line 9).

Afterwards, an iterative group adjustment is performed to reduce load imbalances between the processor groups (line 10). In each iteration step, two groups of processors R_i and R_j are identified such that moving a processor from R_i to R_j reduces the total execution time of the layer while R_i is still large enough to execute all super-tasks assigned to it. The procedure stops when there is no such group left.

5.4.2. Critical-path based scheduling for super-tasks

Critical-path based scheduling algorithms for parallel task graphs consist of two steps: an allocation step that assigns a number of processors to each parallel task and a scheduling step that determines an execution order and maps the parallel tasks onto groups of processors. To adapt critical-path based scheduling algorithms for parallel tasks to the scheduling problem for super-task graphs it is sufficient to modify the allocation step adequately, since this step can guarantee that each super-task is scheduled on a sufficiently large processor group by the scheduling step. Many different heuristics have been proposed for the allocation step. In the following, we consider the allocation steps of the algorithms CPA [29] and CPR [28] and describe their modification for super-tasks.

Both algorithms employ an iterative approach for the allocation step. The iteration starts with an allocation of a single processor to each parallel task. In each step of the iteration, a parallel task is selected and its allocation is increased by one. CPA only considers tasks on the critical path and stops the iteration when the length of the critical path drops below a given threshold. Due to the increased allocation, the critical path may change in each iteration step. CPR on the other hand, first selects a parallel task, increases its processor allocation by one and runs the scheduling step. The change of the allocation is committed if the resulting schedule is better than any previously computed schedule. Otherwise the change is revoked and another parallel task is considered. CPR stops if the currently computed schedule cannot be improved by assigning an additional processor to any task.

Both algorithms only increase the processor allocation of the parallel tasks, i.e., the allocation cannot drop below the initial allocation for any task. To account for the minimum number of processors required by the super-tasks, the initial allocation for each super-task used in the first iteration step has to be greater or equal to the number of CM-tasks included in the respective super-task. Thus, we modify the allocation step by initially assigning exactly m processors to a super-task including m CM-tasks. The modified algorithms resulting from CPA and CPR combined with the transformation of the CM-task graph into the super-task graph and the load balancing from Algorithm 1 are denoted as *CM-CPA* and *CM-CPR*, respectively.

5.5. Building a CM-task schedule

In the final step, the schedule computed for the super-task graph and the allocation functions for the super-tasks are combined into the resulting CM-task schedule. For each super-task, this phase has to determine specific processor groups for the included CM-tasks based on the allocation computed in the load balancing step. Different selections might lead to different re-distribution costs between CM-tasks of different layers. Thus, to reduce communication costs, this step tries to assign CM-tasks connected by a P-relation to the same or at least to overlapping sets of processors. Fig. 7(e) shows the resulting schedule for the example CM-task graph.

6. Experimental evaluation

This section discusses experimental results obtained by applying the scheduling algorithms proposed in the previous section to synthetic CM-task graphs as well as to complex application benchmark programs.

6.1. Simulation results

First, we compare the schedules obtained by the algorithms *CM-Layer*, *CM-CPA*, and *CM-CPR* with *data parallel* and *task parallel* schedules. A data parallel schedule denotes that each super-task of the super-task graph is executed on all available processors, i.e., the individual super-tasks are executed one after another. The assignment of processors to the CM-tasks inside a super-task is computed using the load balancing procedure from Algorithm 1. The task parallel schedule is obtained by assigning a single processor to each CM-task and using a modified list scheduling approach to compute a feasible execution order.

For the simulation, three different test sets of CM-task graphs are used. Each test set comprises 100 different CM-task graphs with the same number of nodes n, $n \in \{10, 100, 1000\}$. The CM-task graphs have been created using an extended version of a graph generation algorithm for directed acyclic graphs [23]. The extended algorithm performs n^2 steps to generate a CM-task graph with n nodes starting with a CM-task graph consisting of n nodes and no edges. In each step, the algorithm selects two nodes at random. If these nodes are connected by a (directed or bidirectional) edge, then this edge is removed from the current graph. Otherwise, the algorithm decides at random whether to connect these two nodes with a directed or with a bidirectional edge. The new edge is only inserted if the resulting CM-task graph remains feasible, i.e., it does not contain cycles of directed edges or conflicting constraints of the execution order defined by the directed and bidirectional edges. At the end, the algorithm inserts an entry node that precedes all nodes without an incoming directed edge and an exit node that succeeds all nodes without an outgoing directed edge.

The parallel execution time of the synthetic CM-tasks is simulated according to the model for parallel tasks used in [24]. This model assumes that each CM-task processes N data elements. The computational complexity $W(A)$ of a CM-task A is either $a \cdot N$ (simulating the processing of a $\sqrt{N} \times \sqrt{N}$ image), $a \cdot N \log N$ (simulating the sorting of an array with N elements), or $a \cdot N^{3/2}$ (simulating the multiplication of two dense $\sqrt{N} \times \sqrt{N}$ matrices), where a is a parameter that is picked uniformly from the interval $[2^6, \ldots, 2^9]$. The parallel execution time of a CM-task A executed on p processors is modeled according to Amdahl's law, i.e., $T(A, p) = \alpha * W(A) + (1-\alpha) * W(A)/p$, where α is the fraction of non-parallelizable code that is picked uniformly from the interval $[0, \ldots, 0.25]$.

Fig. 8 (left) shows the relative performance for different combinations of the number of CM-tasks in the CM-task graph and the number of processors of the platform. The relative performance is computed by first dividing the makespan of the individual schedules obtained by *CM-Layer* by the makespan of the corresponding schedules obtained by the respective algorithm and then computing the average of the resulting values. The figure also shows the minimum and maximum values obtained in the division for each scheduling algorithm. *CM-CPR* is not shown for CM-task graphs with $n = 1000$ nodes, since this algorithm requires several hours or even multiple days to schedule such large graphs due to its high complexity. The results show that *CM-Layer* produces

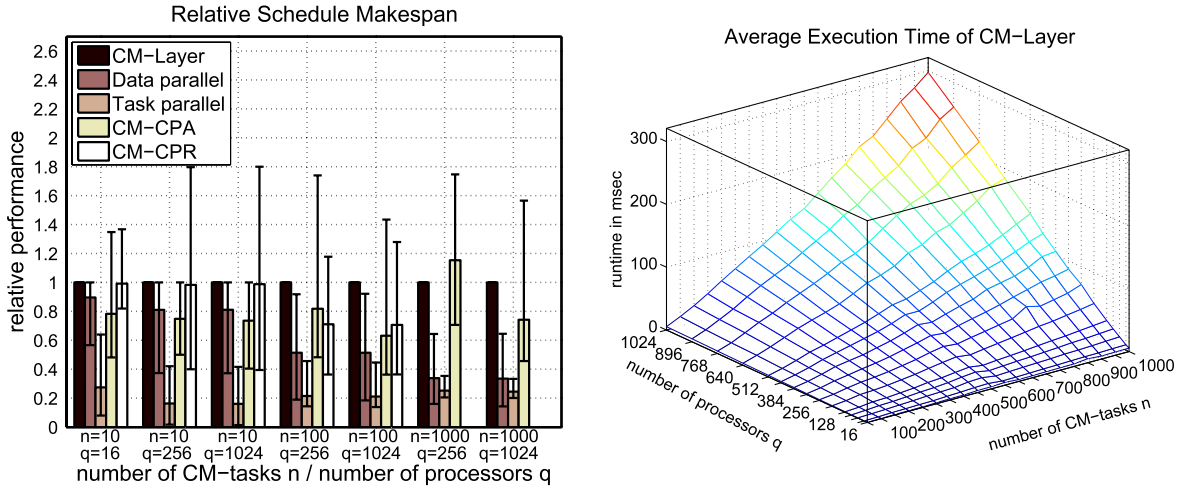

Fig. 8. (Left) Relative performance of different scheduling algorithms in comparison to *CM-Layer*; (right) runtime of the CM-task scheduling algorithm depending on the number of CM-tasks n of the CM-task graph and the number of processors q of the target platform.

Table 1
Overview of the hardware and software configurations of the parallel platforms.

Platform name	CPU type	CPU clock/peak performance	Nodes	Cores per node	MPI library	Interconnection network
CHiC	AMD Opteron 2218 'Santa Rosa'	2.6 GHz/5.2 GFlops/s	530	2 × 2	MVAPICH v1.0	Infiniband 10 GBit/s
JuRoPA	Intel Xeon X5570 'Nehalem'	2.93 GHz/11.72 GFlops/s	2208	2 × 4	ParaStation MPI v5.0	Infiniband 40 GBit/s
SuperMIG	Intel Xeon E7-4870 'Westmere-EX'	2.4 GHz/9.6 GFlops/s	205	4 × 10	IBM MPI v5.2	Infiniband 40 GBit/s

the best schedules on average. In particular, the mixed parallel schedules obtained by *CM-Layer* are always superior to a standard data parallel and a standard task parallel execution. *CM-CPA* and *CM-CPR* are outperformed on average, but may produce up to 80% better results than *CM-Layer* for specific CM-task graphs. This usually happens for very deep CM-task graphs where *CM-Layer* constructs many layers with only a few super-tasks each.

The runtime of *CM-Layer* depicted in Fig. 8 (right) has been measured on an AMD Opteron '"Istanbul"' system clocked at 2.1 GHz. The results show that even large CM-task graphs with 1000 nodes can be scheduled in less than 0.3 s for a platform with 1024 processors. This low execution time makes this algorithm especially suited for the integration into a tool like the CM-task compiler.

6.2. Hardware description

The application benchmarks are executed on three parallel platforms, see Table 1 for an overview of the most important parameters. The Chemnitz High Performance Linux (*CHiC*) cluster consists of 530 nodes, each equipped with two AMD Opteron 2218 dual-core processors clocked with a clock rate of 2.6 GHz. The peak performance of a single core is 5.2 GFlops/s. The nodes are interconnected by an SDR Infiniband network and the MVAPICH 1.0 MPI library is used.

The *JuRoPA* cluster is built up of 2208 nodes, each consisting of two Intel Xeon X5570 (Nehalem) quad-core processors. The processors run at 2.93 GHz leading to a peak performance of 11.72 GFlops/s per core. A QDR Infiniband network connects the nodes and the ParaStation MPI library 5.0 is used.

The *SuperMIG* system consists of 205 nodes, each equipped with four Intel Xeon E7-4870 (Westmere-EX) 10-core processors. The processors are clocked at 2.4 GHz and achieve a peak performance of 9.6 GFlops/s per core. The interconnection is a QDR Infiniband network and the IBM MPI 5.2 library is used.

6.3. Evaluation of the ODE benchmarks

The first set of applications are solvers for systems of ordinary differential equations (ODEs). In particular, we consider the Iterated Runge–Kutta (IRK) method and the Parallel Adams–Bashforth–Moulton (PABM) [37] method, see Fig. 4 for the CM-task specification program of the PABM method. Both methods perform a large number of time steps one after another. Each time step computes a fixed number K of stage vectors. Three different parallel implementations are considered: The *data parallel version* computes the K stage vectors of each time step one after another using all available processors and, thus, contains several global communication operations. The *task parallel version* based on standard *parallel tasks* computes the K stage vectors concurrently on K disjoint equal-sized groups of processors. This restricts the task internal communication to groups of processors but leads to additional global communication for the exchange of intermediate results between the processor groups. An illustration of the task graph for two time steps and $K = 3$ stage vectors is shown in Fig. 1 (b). In the *task parallel version* based on *CM-tasks* a single CM-task computes a specific stage vector over all time steps, see Fig. 1 (c) for an illustration of the CM-task graph. The data exchange between the individual CM-tasks at the end of each time step is organized in communication phases modeled by C-relations and is implemented using orthogonal communication [31]. All program versions are implemented in C and use the MPI library for communication between the processors.

Two different ODE systems have been used for the benchmarks. The first ODE system results from the spatial discretization of the 2D Brusselator equation (BRUSS2D) [14]. The second ODE system arises from a Galerkin approximation of a Schrödinger–Poisson system (SCHROED). The time required to evaluate the entire ODE system depends linearly (BRUSS2D) or quadratically (SCHROED) on the size of the ODE system.

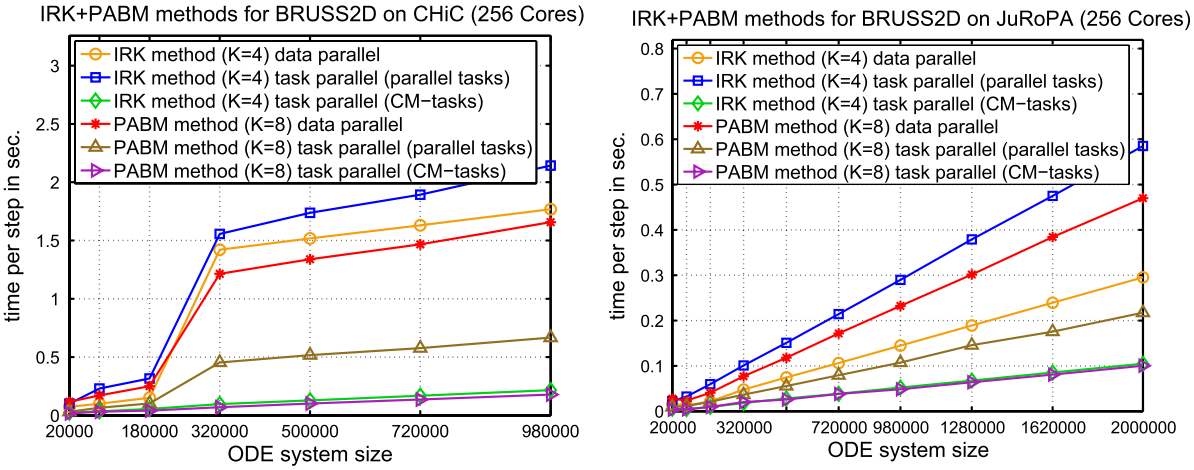

Fig. 9. Measured execution times for a single time step of the IRK method with $K = 4$ stage vectors and the PABM method with $K = 8$ stage vectors on 256 processor cores of the CHiC cluster (left) and the JuRoPA cluster (right).

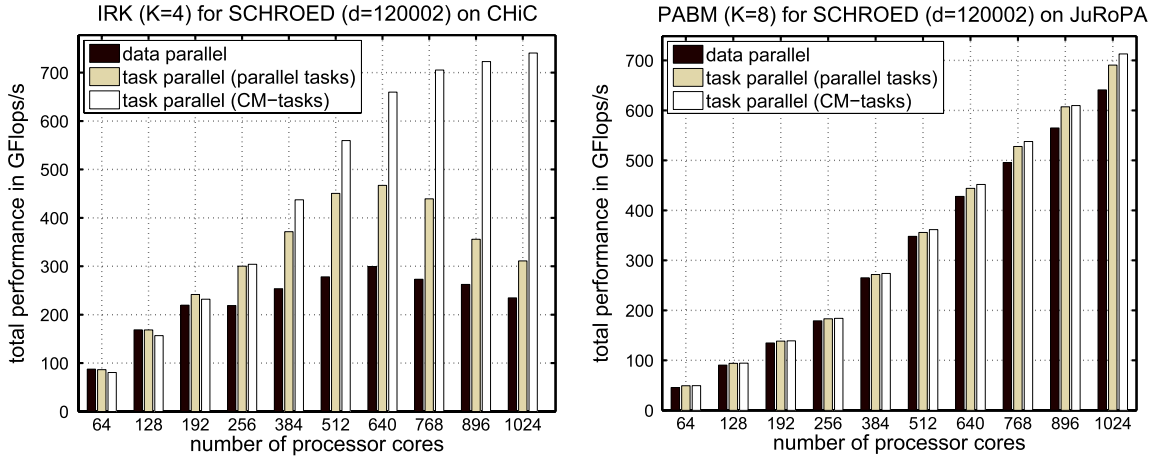

Fig. 10. Performance of the IRK method with $K = 4$ stage vectors on the CHiC cluster (left) and of the PABM method with $K = 8$ stage vectors on the JuRoPA cluster (right) for the SCHROED system.

Fig. 9 shows the average execution times of one time step of the IRK and PABM methods for the BRUSS2D system. The average has been computed 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. The measurements show that a standard data parallel implementation leads to lower execution times compared to standard parallel tasks for the IRK method because additional data re-distribution operations are avoided. For the PABM method, the task parallel version with standard parallel tasks leads to lower runtimes than pure data parallelism because the stage vector computations are decoupled from each other and, thus, much fewer data re-distribution operations are required than for the IRK method. The lowest execution times are achieved by the task parallel version based on CM-tasks for both, the IRK and the PABM method. For example, the runtime of the data parallel implementation of the IRK method on the CHiC cluster can be reduced to one fifth by employing CM-tasks.

Fig. 10 shows the total performance measured for IRK and PABM methods for the SCHROED system. The results show that especially on a large number of processors an efficient organization of the data exchanges as it is provided by the CM-task model is required to obtain a high performance. This is especially true for the CHiC cluster because the interconnection is slower than on the JuRoPA cluster. Compared to a sequential execution, the CM-task implementation achieves a speedup of up to 465 (IRK on 1024 cores of the CHiC cluster) and of up to 790 (PABM on 1024 cores of the JuRoPA cluster).

Fig. 11 (left) compares the performance of the CM-task program version produced by the CM-task compiler with a handwritten CM-task implementation. The relative performance shown in the figure has been obtained by dividing the average execution time of the handwritten version by the runtime of the generated version. The overhead of the generated version is mainly caused by the data re-distribution operations which are implemented by collective communication in the handwritten version and by point-to-point communication in the generated version. The overhead decreases from 20% (for small ODE systems) to under 2% (for large ODE systems) because the share of the coordination code in the total execution time decreases with the system size.

6.4. Evaluation of the NAS benchmarks

The second set of applications is taken from the NAS Parallel Multi-Zone (NAS-MZ) benchmark suite [38]. These benchmarks compute the solution of flow equations on a three-dimensional discretization mesh that is partitioned into zones. One time step consists of independent computations for each zone followed by a border exchange between neighboring zones. The original implementation uses OpenMP for the computations within the zones and MPI for the data exchanges between zones. For the purpose of this article we use a modified version that uses MPI also within the zones and thus allows more flexible scheduling

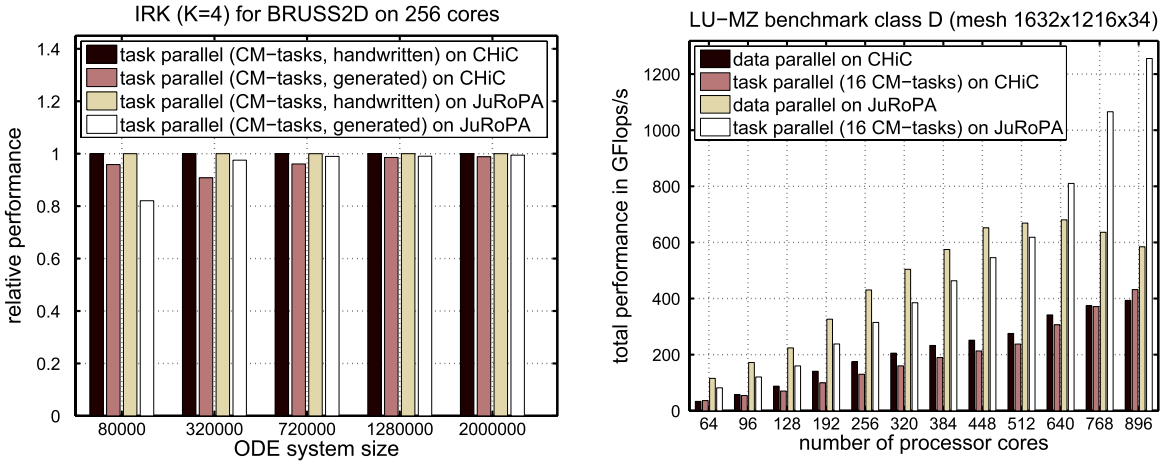

Fig. 11. Relative performance of the generated version of the IRK method compared to a handwritten implementation (left) and performance of the LU-MZ benchmark (right).

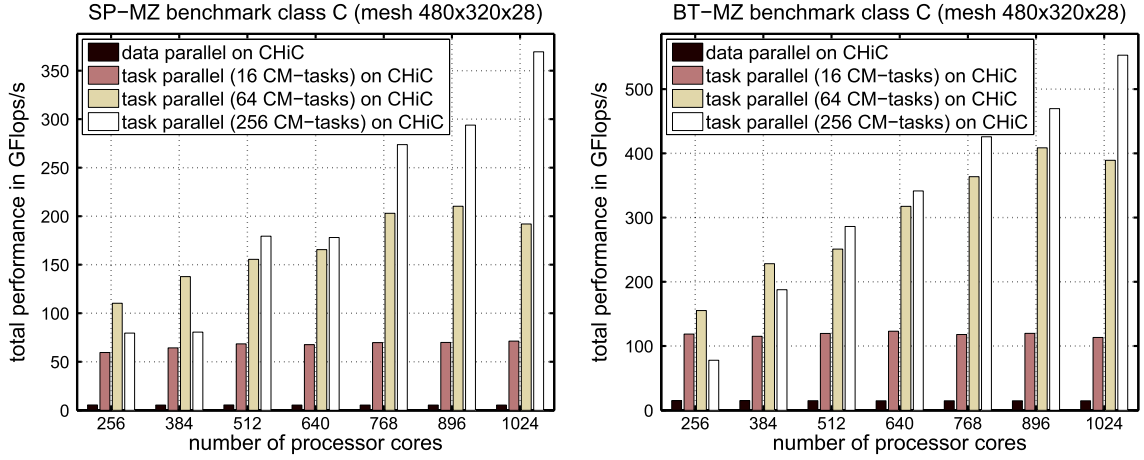

Fig. 12. Performance of the SP-MZ (left) and the BT-MZ (right) benchmarks for different arrangements of zones into CM-tasks.

decisions on cluster systems. Two different implementations are considered: The *data parallel* version uses all processors to compute the individual zones one after another. The *task parallel* version uses a set of CM-tasks each implementing the computations of a subset of the zones and C-relations to model the border exchanges between zones assigned to different CM-tasks. Due to the C-relations all CM-tasks form a single super-task. Thus, the scheduling algorithms *CM-Layer*, *CM-CPA*, and *CM-CPR* compute identical schedules.

Fig. 11 (right) shows the measured performance of the LU-MZ benchmark that consists of 16 equal-sized zones leading to 16 equal-sized processor groups in the CM-task implementation. For a low number of cores, data parallelism leads to a better performance on both platforms due to a better utilization of the cache and the avoidance of communication for the border exchanges. For a high number of cores, the implementation with CM-tasks shows a much better scalability because the number of cores per zone is smaller leading to smaller overall synchronization and waiting times.

Fig. 12 shows the performance of the SP-MZ and BT-MZ benchmarks which both define 256 zones in class C. We compare program versions with 16, 64, and 256 CM-tasks where each CM-task computes 16, 4, and 1 zones one after another, respectively. Data parallel implementations are not competitive for these benchmarks, because the individual zones do not contain enough computations to employ a large number of cores. The number of cores per zone is much smaller in the task parallel versions leading to a much higher performance. In the SP-MZ benchmark all zones have the same size and, thus, equal-sized processor groups are used to execute the CM-tasks. This is only possible when the number of processor cores is a multiple of the number of CM-tasks. In all other cases, load imbalances between the processor groups lead to a degradation of the performance. For example, the program version with 256 CM-tasks has almost the same performance on 512 as on 640 processor cores.

The zones in the BT-MZ benchmark have different sizes and, thus, the assignment of an equal amount of workload to each processor by the load balancing from Algorithm 1 is important. For example, the program version with 256 CM-tasks suffers from load imbalances on 256 cores. These imbalances cannot be eliminated because one core has to be used for each CM-task. On 512 cores, the performance of the program version is approximately 3.7 times higher than on 256 cores. This result indicates that the load imbalances have been reduced considerably.

Next, we compare different super-task scheduling decisions for the SP-MZ and BT-MZ benchmarks. For this purpose, we assign each zone to a separate CM-task and model only a subset of the border exchanges with C-relations. The border exchanges not captured by C-relations are performed by an additional global communication step at the end of each time step. The resulting overhead is small compared to the computation time for a single zone. Depending on the number of C-relations used, different numbers of super-tasks result which are independent of each other.

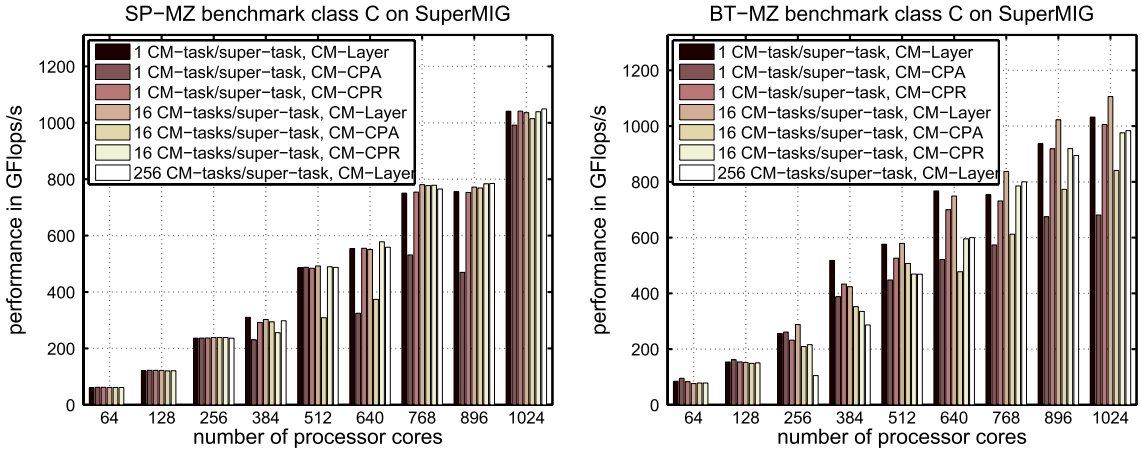

Fig. 13. Performance of the SP-MZ (left) and the BT-MZ (right) benchmarks for different scheduling algorithms.

We consider program versions with 1 CM-task per super-task, i.e., no C-relations at all, 16 CM-tasks per super-task, and 256 CM-tasks per super-task, i.e., the program version with 256 CM-tasks used for the benchmarks in Fig. 12.

Fig. 13 shows the resulting performance for the different scheduling algorithms. Using 256 CM-tasks per super-task leads to a single super-task and, thus, all algorithms compute the same schedule. The results show that the algorithms *CM-Layer* and *CM-CPR* are successful in computing an efficient execution scheme. The schedules produced by *CM-CPA* are competitive in some situations, e.g., for the SP-MZ benchmark executed on 1024 processor cores. But in other cases *CM-CPA* may also deliver schedules that lead to a much lower overall performance, e.g., for the BT-MZ benchmark executed on 1024 processor cores. The reason for this behavior is the decoupling of the allocation step from the scheduling step in *CM-CPA*. As a result, the allocation step may assign a number of processors to the super-tasks that prevents the scheduling step to use all available processors for the execution of independent super-tasks leading to a substantial amount of unused processor time.

7. Related work

There exists a variety of programming models and software tools with the support of mixed task and data parallel applications, see [1,4,8,34] for an overview. The approaches can roughly be classified into language extensions, parallel libraries, and coordination-based approaches. Language extensions are based on an existing programming language with additional annotations or language constructs to describe mixed parallel executions. The basis is usually either a data parallel language that is extended by task parallel constructs, or a task parallel language that is extended with support for data parallelism. Examples for language extensions are Fortran M [11], Opus [5], and Fx [35]. In these approaches, the programmer is responsible for the coordination of the individual data parallel program parts, i.e., the parallel tasks. The dependences between the individual parallel tasks are defined implicitly in the respective source program. In contrast, the CM-task framework includes an explicit specification language that can be transformed to an explicit coordination structure represented in the form of CM-task graphs. The explicit coordination enables a global scheduling, i.e., the adaption to a specific parallel platform can be supported by software tools.

Parallel libraries can support mixed parallel executions by providing suitable library functions, e.g., to coordinate or synchronize data parallel tasks, to manage processor groups and support the execution of data parallel tasks on these groups, or to re-distribute parallel data structures between groups of processors. Examples for such libraries are the HPF/MPI library [12] that allows the coupling of multiple data parallel HPF programs, and the TLib library [33] that provides support for the concurrent execution of parallel tasks on disjoint groups of processors. The TLib approach is especially suited for hierarchical divide-and-conquer algorithms. Similar to the language extensions, the coordination structure is implicit and there is no scheduling support for the entire application.

Coordination-based approaches have an explicit coordination structure that provides a global view on the entire application. Paradigm [30] is a parallelizing compiler, which extracts the parallel task graph from annotations in the program source code. Network of tasks [26] is a programming model in which a parallel application is specified in form of a directed acyclic graph where each graph node represents a parallel program. Both approaches include a scheduler for the mapping of the task graph to different parallel platforms. The interactions between the tasks are restricted to input–output relations. The CM-task model considered in this article is an extension of these approaches which captures additional communication patterns that are modeled by communication relations.

The determination of an optimal schedule for an application consisting of parallel tasks with precedence constraints is an NP-hard problem that is usually solved by scheduling heuristics or approximation algorithms [22]. In this article, we have proposed a layer-based scheduling algorithm that first decomposes a given task graph into layers of independent tasks and then schedules the resulting layers one after another using a list scheduling approach. Many other scheduling algorithms for parallel task use a two-step approach consisting of an allocation step that determines the number of processors for each parallel task and a scheduling step that assigns the parallel tasks to specific sets of processors. The scheduling step is usually based on a modified list scheduling algorithm. The allocation step often uses an iterative approach that starts with an initial allocation (usually one processor per parallel tasks) and repeatedly assigns additional processors with the goal to shorten the critical path of the task graph until a specific stopping criterion is reached. Examples for such algorithms are CPR [28], CPA [29], MCPA [2], Loc-MPS [39] and RATS [18]. These algorithms can easily be extended to the CM-task scheduling problem as described in Section 5.4.2. TSAS [30] defines a convex optimization problem that has to be solved in the allocation step. An alternative to this two step approach is to use evolutionary algorithms for the entire scheduling decision [10,17]. A work stealing approach for the online scheduling of parallel tasks in a shared address space has been investigated in [40]. All of the above mentioned algorithms have been designed for the standard parallel task model and, thus, have to be adapted to additionally consider the C-relations of the CM-task model.

8. Conclusions

In this article, we have presented a parallel programming model with mixed task and data parallelism for coding modular applications. This model is based on parallel tasks where each parallel task can be executed on an arbitrary set of processors and may be hierarchically decomposed into further parallel tasks. Existing programming models for parallel tasks usually consider task graphs with input–output dependences (precedence constraints). We have extended these models by additionally supporting communication between concurrently running parallel tasks. The extended model captures two types of dependences, input–output dependences and communication dependences, thus providing a more flexible way to structure complex modular applications.

The development of applications in the extended model is supported by a compiler framework that transforms a user-provided specification of the task interactions into an executable parallel program. The framework includes a static scheduler that computes a suitable execution scheme based on the characteristics of the target platform. This approach relieves the programmer from the need to specify an explicit task mapping and leads to a portability of the application performance. For the scheduling decision, scheduling algorithms for parallel tasks have to be modified to fit the needs of the extended model. These modifications have been illustrated for layer-based as well as critical-path based scheduling approaches. The overhead of applying the transformation tool is small compared to the performance improvement achievable, thus combining performance efficiency and programmability.

References

[1] H. Bal, M. Haines, Approaches for integrating task and data parallelism, IEEE Concurrency 6 (1998) 74–84.
[2] S. Bansal, P. Kumar, K. Singh, An improved two-step algorithm for task and data parallel scheduling in distributed memory machines, Parallel Computing 32 (2006) 759–774.
[3] K. Barker, K. Davis, A. Hoisie, D. Kerbyson, M. Lang, S. Pakin, J. Sancho, Using performance modeling to design large-scale systems, IEEE Computer 42 (2009) 42–49.
[4] S. Chakrabarti, K. Yelick, J. Demmel, Models and scheduling algorithms for mixed data and task parallel programs, Journal of Parallel and Distributed Computing 47 (1997) 168–184.
[5] B. Chapman, M. Haines, P. Mehrota, H. Zima, J. Van Rosendale, Opus: a coordination language for multidisciplinary applications, Scientific Programming 6 (1997) 345–362.
[6] E. Davidson, T. Cormen, Building on a framework: using FG for more flexibility and improved performance in parallel programs, in: Proceednigs of the 19th IEEE International Parallel and Distributed Processing Symposium, IPDPS'05, IEEE Computer Society, Washington, DC, USA, 2005, pp. 54–63.
[7] J. Dinan, D. Larkins, P. Sadayappan, S. Krishnamoorthy, J. Nieplocha, Scalable work stealing, in: Proceedings of the 21st International Conference on Supercomputing, SC '09, ACM, 2009.
[8] J. Dümmler, T. Rauber, G. Rünger, Mixed programming models using parallel tasks, in: Dongarra, Hsu, Li, Yang, Zima (Eds.), Handbook of Research on Scalable Computing Technologies, Information Science Reference, 2009, pp. 246–275.
[9] J. Dümmler, T. Rauber, G. Rünger, Semi-dynamic scheduling of parallel tasks for heterogeneous clusters, in: Proceedings of the 10th Internationl Symposium on Parallel and Distributed Computing, ISPDC'11, IEEE, 2011, pp. 1–8.
[10] U. Fissgus, Scheduling using genetic algorithms, in: Proceedings of the 20th International Conference on Distributed Computing Systems, ICDCS'00, IEEE, 2000, pp. 662–669.
[11] I. Foster, K. Chandy, Fortran M: a language for modular parallel programming, Journal of Parallel and Distributed Computing 26 (1995) 24–35.
[12] I. Foster, D. Kohr, R. Krishnaiyer, A. Choudhary, Double standards: bringing task parallelism to HPF via the message passing interface, in: Proceedings of the 1996 ACM/IEEE Conference on Supercomputing, SC'96, IEEE, 1996.
[13] M. Frigo, C. Leiserson, K. Randall, The implementation of the cilk-5 multithreaded language, SIGPLAN Notices 33 (1998) 212–223.
[14] E. Hairer, S. Nørsett, G. Wanner, Solving Ordinary Differential Equations I: Nonstiff Problems, second ed., Springer-Verlag, Berlin Heidelberg New York, 1993.
[15] E. Hairer, G. Wanner, Solving Ordinary Differential Equations II: Stiff and Differential-Algebraic Problems, second ed., Springer-Verlag, Berlin Heidelberg New York, 1996.
[16] R. Hoffmann, T. Rauber, Adaptive task pools: efficiently balancing large number of tasks on shared-address spaces, International Journal of Parallel Programming 39 (2011) 553–581.
[17] S. Hunold, J. Lepping, Evolutionary scheduling of parallel tasks graphs onto homogeneous clusters, in: Proceedings of the IEEE Conference on Cluster Computing, CLUSTER '11, IEEE, 2011, pp. 344–352.
[18] S. Hunold, T. Rauber, F. Suter, Redistribution aware two-step scheduling for mixed-parallel applications, in: Proceedings of the 2008 IEEE International Conference on Cluster Computing, CLUSTER '08, IEEE, 2008, pp. 50–58.
[19] M. Kühnemann, T. Rauber, G. Rünger, A source code analyzer for performance prediction, in: Proceedings of the IPDPS '04 Workshop on Massively Parallel Processing, WMPP'04, IEEE, 2004.
[20] M. Kühnemann, T. Rauber, G. Rünger, Performance modelling for task-parallel programs, in: M. Gerndt, V. Getov, A. Hoisie, A. Malony, B. Miller (Eds.), Performance Analysis and Grid Computing, Kluwer, 2004, pp. 77–91.
[21] D. Leijen, W. Schulte, S. Burckhardt, The design of a task parallel library, in: Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA'09, ACM, 2009, pp. 227–242.
[22] J.T. Leung (Ed.), Handbook of Scheduling: Algorithms, Models, and Performance Analysis, CRC Press, Inc., Boca Raton, FL, USA, 2004.
[23] G. Melancon, I. Dutour, M. Bousquet-Melou, Random Generation of Dags for Graph Drawing, Technical Report, INS-R0005, Dutch Research Centre for Mathematics and Computer Science, CWI, Amsterdam, 2000.
[24] T. N'takpé, F. Suter, H. Casanova, A comparison of scheduling approaches for mixed-parallel applications on heterogeneous platforms, in: 6th International Symposium on Parallel and Distributed Computing, IEEE Computer Press, Hagenberg, Austria, 2007.
[25] OpenMP, OpenMP Application Program Interface, Version 3.0, 2008. www.openmp.org.
[26] S. Pelagatti, D. Skillicorn, Coordinating programs in the network of tasks model, Journal of Systems Integration 10 (2001) 107–126.
[27] J. Perez, R. Badia, J. Labarta, A dependency-aware task-based programming environment for multi-core architectures, in: Proceedings of the IEEE International Conference on Cluster Computing, CLUSTER '08, IEEE, 2008, pp. 142–151.
[28] A. Radulescu, C. Nicolescu, A. van Gemund, P. Jonker, CPR: mixed task and data parallel scheduling for distributed systems, in: Proceedings of the 15th International Parallel & Distributed Processing Symposium, IPDPS '01, IEEE, 2001.
[29] A. Radulescu, A. van Gemund, A low-cost approach towards mixed task and data parallel scheduling, in: Proceedings of the International Conference on Parallel Processing, ICPP '01, IEEE, 2001, pp. 69–76.
[30] S. Ramaswamy, S. Sapatnekar, P. Banerjee, A framework for exploiting task and data parallelism on distributed memory multicomputers, IEEE Transactions on Parallel Distributed Systems 8 (1997) 1098–1116.
[31] T. Rauber, R. Reilein, G. Rünger, Group-SPMD programming with orthogonal processor groups, Concurrency and Computation: Practice and Experience 16 (2004) 173–195 (Special Issue on Compilers for Parallel Computers).
[32] T. Rauber, G. Rünger, Compiler support for task scheduling in hierarchical execution models, Journal of Systems Architecture 45 (1998) 483–503.
[33] T. Rauber, G. Rünger, Tlib—a library to support programming with hierarchical multi-processor tasks, Journal of Parallel and Distributed Computing 65 (2005) 347–360.
[34] D. Skillicorn, D. Talia, Models and languages for parallel computation, ACM Computing Surveys 30 (1998) 123–169.
[35] J. Subhlok, 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, ACM Press, 1997, pp. 1–12.
[36] E. Tejedor, M. Farreras, D. Grove, R. Badia, G. Almasi, J. Labarta, ClusterSs: a task-based programming model for clusters, in: Proceedings of the 20th International Symposium on High Performance Distributed Computing, HPDC'11, ACM, 2011, pp. 267–268.
[37] P. van der Houwen, E. Messina, Parallel Adams methods, Journal of Computational and Applied Mathematics 101 (1999) 153–165.
[38] R. van der Wijngaart, H. Jin, The NAS Parallel Benchmarks, Multi-Zone Versions, Technical Report, NAS-03-010, NASA Ames Research Center, 2003.
[39] N. Vydyanathan, S. Krishnamoorthy, G. Sabin, Ü. Çatalyürek, T. Kurç, P. Sadayappan, J. Saltz, An integrated approach to locality-conscious processor allocation and scheduling of mixed-parallel applications, IEEE Transactions of the Parallel Distributed Systems 20 (2009) 1158–1172.
[40] M. Wimmer, J. Träff, Work-stealing for mixed-mode parallelism by deterministic team-building, in: Proceedings of the 23rd ACM Symposium on Parallelism in Algorithms and Architectures, SPAA'11, ACM, 2011.

Jörg Dümmler received a Doctoral degree in computer science from the Chemnitz University of Technology in 2010. Since then, he has been working as a postdoctoral researcher at the Chemnitz University of Technology. His current research interests include high-level parallel programming models, mixed parallel executions, and the scheduling of parallel tasks.

Thomas Rauber received a Ph.D. and habilitation degree in computer science from the University des Saarlandes (Saarbrücken) in 1990 and 1996, respectively. From 1996 to 2002, he has been professor for computer science at the Martin-Luther-University Halle–Wittenberg. He joined the University Bayreuth in 2002 where he now holds the chair for parallel and distributed systems. His research interests include parallel and distributed algorithms, programming environments for parallel and distributed systems, compiler optimizations and performance prediction.

Gudula Rünger received a Doctoral Degree in Mathematics from the University of Cologne in 1989 and a Habilitation in Computer Science from the Saarland University, Saarbrücken, in 1996. From 1997 to 2000 she has been a Professor of Parallel Computing and Complex Systems at the University Leipzig. Since 2000 she is a full Professor of Computer Science at Chemnitz University of Technology. Her research interests include parallel and distributed programming, parallel algorithms for scientific application as well as software development in science and engineering.