Proceedings of ALGORITMY 2012
pp. 261–271

INTERACTION PATTERNS FOR CONCURRENTLY EXECUTED PARALLEL TASKS

JÖRG DÜMMLER*

Abstract. Large parallel applications often consist of multiple program parts that can be implemented separately in form of parallel modules. The interactions between these modules can be captured by a high-level coordination structure. This coordination structure represents the data flow within the parallel application and defines restrictions of the execution order of the parallel modules. The coordination structure of an application can be defined independently of the modules using a high-level specification language.

This article focuses on possible interactions between concurrently executed parallel modules that have to exchange data during their execution. For this purpose, several interaction patterns are considered and their definition in the specification language is illustrated. An experimental evaluation using different application benchmarks underlines the feasibility of the programming approach and shows the use of the interaction patterns.

Key words. parallel module, parallel task, pattern, programming model

AMS subject classifications. 68Q85,68U01

1. Introduction.

Parallel programming models based on parallel modules lead to efficient implementations for many large application programs. In these programming models, an application is decomposed into a set of parallel modules where each module is a parallel code that can run on multiple processors of a parallel target platform. A coordination structure specifies how the parallel modules of an application interact with each other and which dependencies have to be considered for their execution. For the coordination structure, directed acyclic graphs [8, 15], Macro Dataflow Graphs (MDGs) [9], and series-parallel (SP) graphs [11] have been used.

These programming models provide several advantages. First, the execution of such an application can be adapted to the computation and communication performance of the parallel target platform. Some of the modules of an application may require an execution one after another due to data or control dependencies, but other modules may also be independent of each other and, thus, allow a flexible execution order, i.e., both, an execution one after another and a concurrent execution on disjoint sets of processors are possible. In the case of a concurrent execution, also the number of processors assigned to each module may be adjusted accordingly. Thus, the portability of the application performance is increased. Second, the overall communication overhead of the entire application can often be reduced on platforms with a distributed memory, since communication and synchronization operations within the parallel modules can be restricted to subsets of processors. Third, the specification of an application can be separated from its implementation on a specific parallel platform. The programmer only needs to specify the high-level interactions between the parallel modules and a software tool or the runtime system is responsible for selecting a suitable scheduling and mapping to the target platform. Thus, the programmer is relieved from many low-level implementation details, such as the management of the

*Chemnitz University of Technology, Department of Computer Science, 09111 Chemnitz, Germany (djo@cs.tu-chemnitz.de).

data flow within the application.

Most programming models only support input-output interactions between parallel modules. This article focuses on the programming model of communicating multiprocessor tasks (CM-tasks) which has first been proposed in [4]. The CM-task programming model additionally supports an interaction between parallel modules during their execution, e.g., to exchange intermediate results. As a result, the CM-task model allows a more flexible structuring of an application into modules which is especially beneficial for time stepping method that require a global data exchange at the end of each time step. The CM-task model supports several patterns for the interaction between concurrently executed parallel modules. This article describes a selection of these patterns that often occur in parallel applications and shows their specification for a programming tool that supports the development of CM-task programs.

The rest of this article is organized as follows. Section 2 describes the CM-task programming model in detail and Sect. 3 presents a corresponding programming support tool. Section 4 discusses possible interaction patterns for concurrently executed modules and shows their specification. An experimental evaluation is presented in Sect. 5. Section 6 discusses related work and Sect. 7 concludes the article.

2. Programming model. A parallel CM-task application is composed of a set of CM-tasks where each CM-task implements a subset of the computations of the application. A CM-task is a parallel module that can be executed on a flexible number of processors. Each CM-task has a specific interface that defines a set of input parameters that are required for the execution of the CM-task, a set of output parameters that are produced by the CM-task, and a set of communication parameters that are exchanged with concurrently executed CM-tasks. Internally, a CM-task may comprise communication and synchronization operations that are performed by the processors assigned to the CM-task for execution. Additionally, a CM-task may also comprise external communication operations that are used to exchange data with other CM-tasks that are executed concurrently on a disjoint set of processors.

The programming model distinguishes basic and composed CM-tasks. Basic CM-tasks are provided by the application developer and are not decomposed further. These can be parallel functions or library routines, e.g., for the multiplication of two matrices. Composed CM-tasks consist of activations of other CM-tasks and a coordination structure that defines the interactions between these CM-task activations. The interactions are captured by two types of relations: precedence relations (P-relations) and communication relations (C-relations). A P-relation exists between two CM-tasks A and B if A produces an output parameter that is required as an input for B. In this case, A and B have to be executed one after another. Moreover, in the case of a distributed memory target platform, a data re-distribution operation may be required between the execution of A and B. Such an operation is necessary if A and B are executed on different sets of processors or if the output data

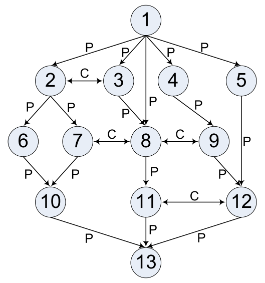


FIG. 2.1. *Example for a CM-task graph with bidirectional edges representing communication relations (annotation C) and directed edges representing precedence relations (annotation P).*

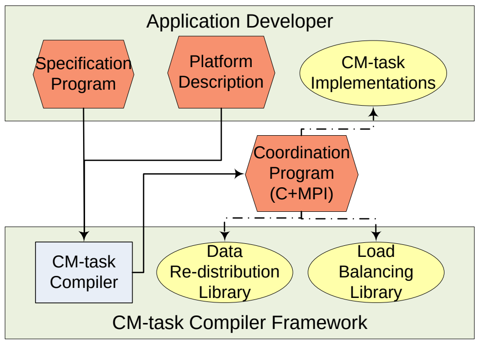


FIG. 3.1. *Overview of the CM-task framework.*

distribution of A and the input data distribution of B do not match. A C-relation exists between two CM-tasks A and B if A and B communicate with each other during their execution, e.g., to exchange intermediate results. In this case, A and B have to be executed concurrently on disjoint sets of processors. Independent CM-tasks, i.e., CM-tasks not connected by a P-relation or a C-relation, can be executed one after another or concurrently.

A composed CM-task can be represented by a CM-task graph $G = (V, E)$ where the set V of nodes represents the CM-task activations. The set E of edges comprises two disjoint subsets E_P and E_C with $E = E_P \cup E_C$ and $E_P \cap E_C = \emptyset$. E_P contains directed edges representing the P-relations and E_C contains bidirectional edges representing the C-relations between the CM-task activations. An example for a CM-task graph is shown in Fig. 2.1.

3. Programming support. This sections discusses the CM-task framework that provides tool support for the development of CM-task programs.

3.1. CM-task framework. The CM-task framework provides support for the specification of CM-task applications, the transformation of an application specification into a platform-specific implementation, and runtime support for the execution of CM-task applications. An overview of the CM-task framework is shown in Fig. 3.1.

For the framework, the user has to provide

- the high-level interactions between the CM-tasks of the application in form of a platform-independent specification program, see Subsect. 3.2 for a detailed description of the underlying specification language;
- the properties of the parallel target platform, such as the number of available processors and the communication and computation performance;
- implementations of the basic CM-tasks in form of parallel functions.

The CM-task compiler transforms the user-provided application specification and platform description into an executable coordination program that is adapted to the target platform specified. The transformation process includes the construction of the CM-task graph from the specification program, the scheduling of the CM-task graph for the target platform, the insertion of the required data re-distribution operations, and the generation of the final coordination program. The coordination program produced is responsible for the execution of the CM-tasks on the processor sets defined by the computed schedule, the execution of the data re-distribution operations resulting from P-relations between CM-tasks using a runtime library, and the provision

```
$M \rightarrow$ 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. 3.2. *Grammar for the specification of the available task parallelism within a composed CM-task (simplified).*

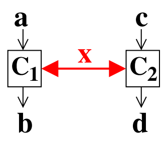
of a common communication context for CM-tasks connected by C-relations. The execution of the data transfers along the C-relations is not part of the coordination program and has to be implemented inside the user-provided parallel functions.

The CM-task compiler supports two modes for the generation of the coordination program. In the static mode, the scheduling is done entirely at compile-time leading to a more optimized coordination program. In the semi-dynamic mode, only the execution order of the CM-tasks is fixed at compile time. The processor sets used to execute the CM-tasks are flexible and can be adapted at runtime by the load balancing library provided within the CM-task framework.

3.2. Specification language. CM-task specification programs describing the platform-independent details of a parallel application are provided in the CM-task specification language. The language supports definitions of constants, data types, e.g., multidimensional arrays, data distribution types, e.g., block-cyclic data distributions over multidimensional processor meshes, basic CM-tasks, and composed CM-tasks. The definition of a basic CM-task consists of the interface and cost information. The interface defines the input, output and communication parameters along their associated data types and data distribution types. The cost information is provided in form of a symbolic formula depending on the number of executing processors and platform-specific parameters that are provided independently in a separate platform description input file.

The definition of a composed CM-task consists of an interface as for basic CM-tasks, a list of local variables used to store intermediate results, and a hierarchical dependence expression, see Fig. 3.2 for the underlying grammar. The dependence expression consists of activations of basic and composed CM-tasks that are defined using the name of the CM-task and a suitable parameter list, and a variety of operators to define possible execution orders of these CM-task activations. The operators `cpar` and `cparfor` define a concurrent execution of multiple CM-tasks. These CM-tasks are connected by C-relations if they share a common communication parameter. The operators `seq`, `for`, and `while` define a consecutive execution of program parts. In this case, P-relations may exist between these program parts. Independent program parts are defined using the operators `par` and `parfor`. In this case, no interaction between these parts in form of P-relations or C-relations is allowed. The conditional execution of program parts can be defined using the operator `if`.

4. Interaction patterns. This section presents several patterns for the interaction between concurrently executed CM-tasks. A specification of an entire application may contain several of these patterns embedded in the specification program.

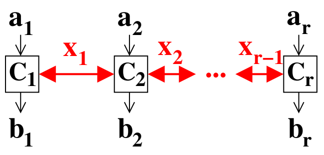


```
cpar {
 C1(a/* in */, b/* out */, x/* comm */);
 C2(c/* in */, d/* out */, x/* comm */);
}
```

FIG. 4.1. *(Left) Fragment of a CM-task graph for the point-to-point communication pattern. (Right) Corresponding fragment of the CM-task specification program. The access modes of the parameters (in, out, comm) are given as comments.*

4.1. Point-to-point pattern. The basic communication pattern is an interaction of two CM-tasks that are executed concurrently by disjoint sets of processors. This kind of communication can be used to provide input data or exchange intermediate results in large multi-disciplinary applications combining algorithms from different fields, e.g., aircraft design or environmental simulations. This communication pattern captures both, a directed data transfer from a CM-task A to a CM-task B as well as a bidirectional data exchange where A transmits data to B and vice versa. Both cases are modeled by a C-relation between the respective CM-tasks. The point-to-point pattern is specified by combining the CM-task activations using the operator `cpar` and providing a common communication parameter to these CM-tasks. Figure 4.1 shows an illustration and an example specification fragment for this pattern.

4.2. Pipeline pattern. The pipeline pattern uses a fixed number of stages where each data element has to pass all pipeline stages one after another. This pattern occurs, for example, in stream-based applications, such as programs processing data periodically produced by sensors [12], image analysis programs, and graphics renderers where autonomous filters have to be applied [6]. For a specific data element an interaction is required between successive pipeline stages. The stages have to be executed consecutively for a specific element, but a concurrent execution is possible for different data elements.

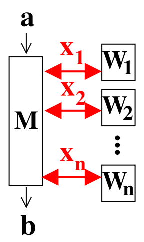


```
cpar {
 C1(a1/* in */,b1/* out */,x1/* comm */);
 C2(a2/* in */,b2/* out */,x1,x2/* comm */);
 C3(a3/* in */,b3/* out */,x2,x3/* comm */);
 ...
 Cr(ar/* in */,br/* out */,xr1/* comm */);
}
```

FIG. 4.2. *(Left) Fragment of a CM-task graph showing a pipeline with r stages. (Right) Corresponding fragment of the CM-task specification program.*

In the CM-task model, each pipeline stage is represented by a separate CM-task. The required data transfers between neighboring stages are modeled by a C-relation between the respective CM-tasks. In the specification program, a pipeline can be specified by combining the CM-tasks representing the pipeline stages with the operator `cpar`. The data transfers between neighboring stages are specified by providing a suitable communication parameter to the respective CM-tasks. As a consequence, each CM-task (except the CM-tasks representing the first and the last pipeline stage) needs two communication parameters. An example specification and the corresponding fragment of the CM-task graph are shown in Fig. 4.2.

4.3. Master/worker pattern. In the master/worker pattern, there is a distinguished master that coordinates the execution and distributes work units to a set of identical workers. Each worker carries out the computations assigned independent of the other workers and delivers the results back to the master. In hierarchical applications, each processed piece of work can introduce new work units. In this case, the master can utilize a task pool to manage the currently available work units. Usually, the master and each of the workers are executed by a single processor. But it might also be beneficial to support an execution of the workers on multiple processors if the number of available processors exceeds the average number of work units in the task pool or the underlying algorithm has a high memory requirement.



```
cpar {
  M(a/* in */, b/* out */,
        x[1],...,x[n]/* comm */);
  cparfor (i=1:n) { W(x[i]/* comm */); }
}
```

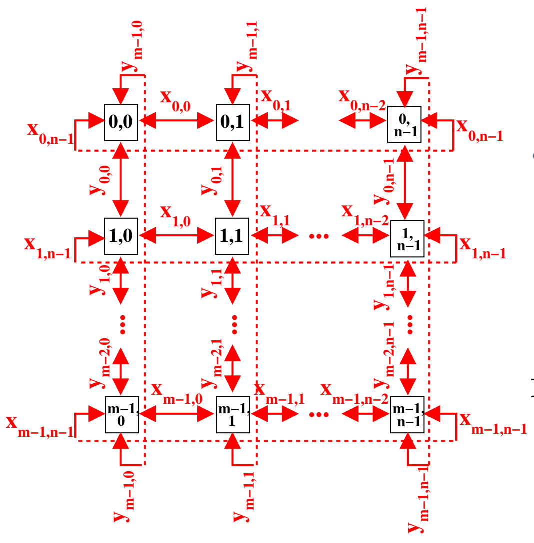
FIG. 4.3. *(Left) Fragment of a CM-task graph for the master/worker pattern. (Right) Corresponding fragment of the CM-task specification program.*

In the CM-task model, the master and each of the n workers are represented by a separate CM-task, leading to $n+1$ CM-tasks for this pattern. There is a C-relation between the master and each of the workers, leading to a total of n C-relations. There is no communication between different workers and, therefore, there are no relations between the respective CM-tasks. Figure 4.3 shows an illustration of the pattern along with an example specification.

4.4. Mesh pattern. In the mesh pattern, a set of CM-task is aligned in a multi-dimensional mesh where each CM-task communicates with its neighbors, see Fig. 4.4 (left) for an illustration. This pattern can be useful for domain decomposition methods that partition the global discretization mesh into a set of partially overlapping zones. Between adjacent zones, an exchange of border values may be required for these methods. Examples for such applications are flow solvers as provided by the NAS Multi-zone benchmarks [13], see Sect. 5.2 for benchmark results.

Figure 4.4 shows the CM-task graph fragment and a suitable CM-task specification for a two-dimensional mesh. The input and output parameters have been omitted to improve readability. Each CM-task has four communication parameters, since there are four neighbors in the two-dimensional case. Each of the communication parameters of a specific CM-task is shared with exactly one neighbor. In the specification example, the communication parameter `x` is used to connect horizontal neighbors and the communication parameter `y` connects vertical neighbors.

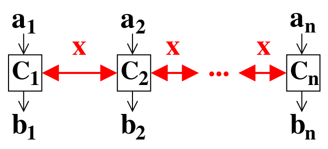
4.5. Collective pattern. The patterns discussed previously are restricted to data exchanges between pairs of CM-tasks. The CM-task model also supports an interaction between more than two concurrently executed CM-tasks, i.e., a collective interaction. Such an interaction is necessary when an intermediate result produced by



```
cparfor (i=0:m-1) {
 cparfor (j=0:n-1) {
  C(i, j/* in */,
    x[i][j]/* comm, right */,
    y[i][j]/* comm, bottom */,
    x[i][(j-1)%n]/* comm, left */,
    y[(i-1)%m][j]/* comm, top */);
 }
}
```

FIG. 4.4. *(Left) Illustration of the mesh pattern using a two-dimensional $m \times n$ mesh. (Right) Corresponding fragment of the CM-task specification program.*

a CM-task is required by multiple other CM-tasks or has to be made available globally. An example are the orthogonal communication operations that exist in many solvers for ordinary differential equations [10], see also the application benchmarks considered in Subsect. 5.1.



```
cparfor (i=1:n) {
  C(i, a[i]/* in */,
    b[i]/* out */,
    x/* comm */);
}
```

FIG. 4.5. *(Left) Fragment of the CM-task graph for a collective interaction of n CM-tasks. (Right) Corresponding fragment of the CM-task specification program.*

Figure 4.5 illustrates the specification of a collective interaction of n identical CM-tasks and the resulting fragment of the CM-task graph. If the CM-tasks are not identical the operator `cpar` instead of `cparfor` has to be used to specify this pattern. In both cases, all CM-tasks participating in the collective data exchange have to define the same communication parameter.

5. Experimental evaluation. This section evaluates the collective pattern and the mesh pattern using appropriate application benchmarks. The benchmarks have been executed on the Chemnitz High Performance Linux (CHiC) cluster. This cluster is built up of 530 nodes each comprising two AMD Opteron 2218 dual-core processors 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 a 10 GBit/s Infiniband network. MVAPICH 1.0.3 has been used as an MPI library and the Pathscale Compiler 3.1 with full optimizations has been used to compile the benchmark programs.

5.1. Evaluation of the collective pattern. There exist several solvers for systems of ordinary differential equations (ODEs) that perform a large number of time

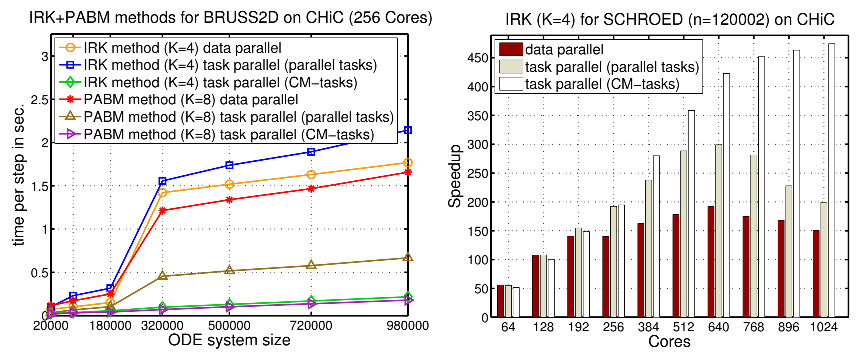


FIG. 5.1. *Benchmark results for ODE solvers using the collective communication pattern.*

steps one after another where each time step includes the evaluation of a fixed number K of stage vectors. The computations for these vectors are usually not independent from each other, but require an exchange of intermediate results between all K vectors. These data exchanges can be implemented using orthogonal communication to reduce the communication overhead [10]. Examples for such solvers are Iterated Runge-Kutta (IRK) methods and Parallel Adams-Bashforth-Moulton (PABM) methods. In the CM-task model, each stage vector can be computed by a separate CM-task and the data exchanges can be modeled using appropriate C-relations according to the collective interaction pattern, see Subsect. 4.5.

For the benchmarks, three different implementations are considered. The data parallel version computes the K stage vectors one after another using all processors. The task parallel version with parallel tasks computes the K vectors concurrently and exchanges the intermediate results using global communication. This corresponds to an implementation in a model that only supports input-output relations. The task parallel version with CM-tasks also computes all vectors concurrently, but uses orthogonal communication for the data exchanges. Figure 5.1 (left) shows the average execution times for a single time step for an ODE system that results from the spatial discretization of the 2D Brusselator equation (BRUSS2D) [5]. The results show a tremendous decrease of the execution time by using the optimized communication pattern supported by CM-tasks. Figure 5.1 (right) shows the speedups for an ODE system resulting from the Galerkin approximation of a Schrödinger-Poisson system (SCHROED). The CM-task implementation shows a much better scalability than the other implementation variants.

5.2. Evaluation of the mesh pattern. The NAS-MZ parallel benchmarks [13] provide several solvers for flow equations defined over a three-dimensional discretization mesh that is partitioned into zones. One time step of these solvers first performs independent computations for each zone followed by a border exchange between neighboring zones. In the CM-task model, one zone or multiple neighboring zones can be implemented by a CM-task. The resulting CM-tasks then interact with each other in form of a mesh, see Subsect. 4.4.

The LU-MZ benchmark defines a 4×4 mesh of equal-sized zones. Figure 5.2 (left) compares the overall performance of a pure data parallel implementation that computes all zones one after another using all available processor and a CM-task implementation where each zone is implemented by a separate CM-task. Benchmark classes C (global mesh size $480 \times 320 \times 28$) and D (global mesh size $1632 \times 1216 \times 34$)

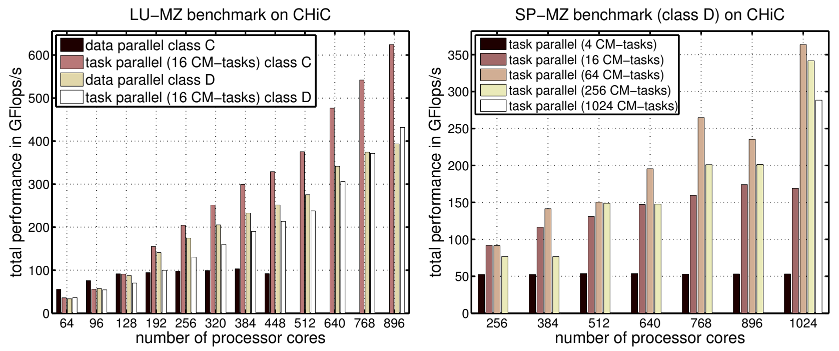


FIG. 5.2. *Benchmark results for different implementation variants of the LU-MZ benchmark (left) and the SP-MZ benchmark (right) on the CHiC cluster.*

have been used. The results show that the data parallel version is more efficient on a low number of processors, since the border exchanges can be performed locally without communication. But for a high number of processors, a concurrent computation of the individual zones on disjoint sets of processors as supported by the CM-task model is required for a high performance.

The SP-MZ benchmark class D defines a 32×32 mesh of equal-sized zones. Figure 5.2 (right) compares the performance for different numbers of CM-tasks. The results show that an implementation with 64 CM-tasks where each CM-tasks computes 16 zones leads to the highest performance.

6. Related work.

Several programming models based on parallel modules have been proposed. Paradigm [9] is a parallelizing compiler that extracts a task graph from a source program with additional annotations, schedules the task graph for a specific parallel platform and produces a parallel implementation as an output. The TwoL system [11] includes a specification language to define the available degree of task parallelism of an application and uses a transformation-based approach to translate a given specification into a platform-specific implementation. In contrast to the CM-task model, communication relations between concurrently executed parallel modules are not supported by these approaches.

Interaction patterns between parallel program fragments can also be specified using skeletons. P3L [7] is a coordination language for skeletons that supports both, data parallel patterns, such as map, reduce and scan as well as task parallel patterns, such as pipelines and task farms. TaskHPF [1] includes a special language to describe the high-level task parallel interactions. The available skeletons allow the definition of pipeline-based computations and the replication of pipeline stages with a low scalability. LLC [3] extends *C* with OpenMP-like annotations to define different patterns, such as pipelines and master/worker schemes. ASSIST [14] is a framework for the composition of sequential and parallel modules to complex applications. The interactions between the modules are described in a coordination language in form of a task graph. SBASCO [2] is a pattern-based coordination language to define domain decomposition methods. It supports the multiblock-pattern for the definition of different domains that interact with each other using border exchanges, the pipeline-pattern, and the replicate-pattern to define multiple instances of a specific program part. In contrast to the CM-task model, there are special operators for each supported pattern. To support additional patterns, the input language has to be extended accordingly.

In contrast, the operators of the CM-task specification language only define possible execution orders of program fragments. The concrete interactions are defined using suitable input, output, and communication parameters. Thus, a large variety of different patterns can be defined without modifying the language or the corresponding programming support tools.

7. Conclusion. This article has discussed the programming model of communicating multiprocessor tasks (CM-tasks). This model supports two types of interactions between parallel modules: precedence relations resulting from input-output dependencies and communication relations resulting from an interaction between parallel modules during their execution. The development of CM-task programs is supported by the CM-task framework that transforms a platform-independent specification of the application structure into a platform-specific implementation based on the characteristics of the target platform.

The article has also discussed several different patterns for the interaction between CM-tasks connected by communication relations. In particular, patterns for a point-to-point interaction, for pipeline-based computations, for master/worker parallelization approaches, for mesh-based computations, and for collective interactions have been presented. For each pattern, a corresponding CM-task specification has been shown. The experimental evaluation has indicated that CM-task implementations can result in a higher parallel performance compared to other execution schemes.

REFERENCES

[1] S. Ciarpaglini, L. Folchi, S. Orlando, S. Pelagatti, and R. Perego. Integrating Task and Data Parallelism with taskHPF. In *Proc. of the International Conference on Parallel and Distributed Techniques and Applications (PDPTA '00)*, pages 2485–2491. CSREA Press, 2000.

[2] M. Díaz, B. Rubio, E. Soler, and J.M. Troya. SBASCO: Skeleton-Based Scientific Components. In *Proc. of the 12th Euromicro Workshop on Parallel, Distributed and Network-Based Processing (PDP '04)*, pages 318–325. IEEE, 2004.

[3] A.J. Dorta, P. López, and F. de Sande. Basic Skeletons in llc. *Parallel Computing*, 32(7-8):491–506, 2006.

[4] J. Dümmler, T. Rauber, and G. Rünger. Communicating Multiprocessor-Tasks. In *Proc. of the 20th International Workshop on Languages and Compilers for Parallel Computing (LCPC '07)*, volume 5234 of *LNCS*, pages 292–307. Springer-Verlag, 2007.

[5] E. Hairer, S.P. Nørsett, and G. Wanner. *Solving Ordinary Differential Equations I: Nonstiff Problems*. Springer-Verlag, Berlin Heidelberg New York, 2nd edition, 1993.

[6] J.L. Ortega-Arjona. *Patterns for Parallel Software Design*. Wiley Publishing, 1st edition, 2010.

[7] S. Pelagatti and D.B. Skillicorn. Coordinating Programs in the Network of Tasks Model. *Journal of Systems Integration*, 10(2):107–126, 2001.

[8] A. Radulescu, C. Nicolescu, A.J.C. van Gemund, and P. Jonker. CPR: Mixed Task and Data Parallel Scheduling for Distributed Systems. In *Proc. of the 15th International Parallel & Distributed Processing Symposium (IPDPS '01)*, pages 39–46. IEEE, 2001.

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

[10] T. Rauber, R. Reilein-Ruß, and G. Rünger. Group-SPMD Programming with Orthogonal Processor Groups. *Concurrency and Computation: Practice and Experience, Special Issue on Compilers for Parallel Computers*, 16(2-3):173–195, 2004.

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

[12] 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.

[13] R.F. van der Wijngaart and H. Jin. The NAS Parallel Benchmarks, Multi-Zone Versions. Technical Report NAS-03-010, NASA Ames Research Center, 2003.

[14] M. Vanneschi. The Programming Model of ASSIST, an Environment for Parallel and Distributed Portable Applications. *Parallel Computing*, 28(12):1709–1732, 2002.

[15] N. Vydyanathan, S. Krishnamoorthy, G. Sabin, Ü.V. Çatalyürek, T.M. Kurç, P. Sadayappan, and J.H. Saltz. An Integrated Approach to Locality-Conscious Processor Allocation and Scheduling of Mixed-Parallel Applications. *IEEE Transactions on Parallel and Distributed Systems*, 20(8):1158–1172, 2009.