A Transformation Framework for Communicating Multiprocessor-Tasks

Jörg Dümmler
Chemnitz University of Technology
Department of Computer Science
djo@cs.tu-chemnitz.de

Thomas Rauber
Bayreuth University
Angewandte Informatik II
rauber@uni-bayreuth.de

Gudula Rünger
Chemnitz University of Technology
Department of Computer Science
ruenger@cs.tu-chemnitz.de

Abstract

Parallel programming models based on a mixture of task and data parallelism have shown to be successful in addressing the increasing communication overhead of distributed memory platforms with a large number of processors. In these models, an application is decomposed into a set of parallel tasks that can run on an arbitrary number of processors. The communication between different tasks is allowed only at the start and the end of a task, thus limiting the possible communication patterns and the potential granularity of the tasks.

In this paper, we consider an extended parallel programming model that additionally supports communication between running parallel tasks. We describe a specification language for applications in the new programming model and propose a transformation framework for a step-wise derivation of an executable message passing program from the specification language. The advantages of the approach are demonstrated for solution methods for ordinary differential equations.

1 Introduction

The scalability of data parallel scientific applications is often limited by the communication overhead that increases with the number of executing processors. A possible approach to overcome this limitation is to use a programming model based on a combination of task and data parallelism [3]. In these models, a parallel application is subdivided into a set of parallel tasks that are often called multiprocessor tasks, malleable tasks or moldable tasks; for the purpose of this paper we use the short term M-tasks. Each M-task processes a different part of the application and can be executed by an arbitrary number of processors. The internal communication overhead of an M-task can therefore be kept low by running on a subset of the available processors. To capture the computing power of all processors, multiple M-tasks can be executed in parallel on disjoint processor groups. M-tasks have input and output parameters; an M-task can produce output parameters that serve as an input of other M-tasks. Thus, dependencies between M-tasks arise that limit the possible execution orders of the tasks. These dependencies are represented by a coordination structure that is usually given in the form of a directed acyclic graph (dag) or a series-parallel graph (SP-graph). An example is the TwoL (two level) model[14].

The communication between M-tasks is restricted to the beginning or the end of their execution, thus limiting the possible granularity of the M-tasks. For example, in time stepping methods that require a data exchange at the end of each time step, M-tasks are limited to execute a single time step. As a consequence, each time step involves the overhead of starting the appropriate M-tasks and supplying their input parameters in the correct data distribution. A better way to structure such an application is to use tasks that keep running over all time steps and use orthogonal communication to exchange data in each step.

In this paper, we introduce the model of communicating M-tasks (CM-tasks) that comes as a natural extension to existing M-task programming models. The CM-task model additionally incorporates communication between running tasks and is therefore able to represent more complex communication patterns such as orthogonal communication. This also leads to a more complex coordination structure, the CM-task graph, that describes possible execution orders of the included CM-tasks. In general, many different execution schemes, i.e. schedules, are possible. Which schedule achieves the best results depends on the communication and computation performance of the target platform and on the structure of the application itself.

To support the development of parallel applications in the CM-task programming model, we propose a specification language and a transformation framework. The specification language allows the definition of hierarchical CM-task applications. The transformation framework generates an executable coordination program based on the MPI message passing library by employing a number of intermediate transformation steps. These steps include an analysis of the dataflow, scheduling and load balancing methods, inserting communication operations and generating the output. The framework supports a static compilation approach for homogeneous target platforms and a semi-dynamic approach that enables the generated program to adapt to the target platform.

This paper is organized as follows. Section 2 explains the programming model of CM-tasks, Section 3 introduces the specification language for CM-task applications and presents an example. The transformation framework is described in detail in Section 4. Section 5 presents runtime results for the example application. Related work is discussed in Section 6 and Section 7 concludes the paper.

2 Programming model

In the CM-task programming model, a parallel application is represented by a set of communicating M-tasks (CM-tasks), which exhibit an internal computational structure that allows an execution on an arbitrary number of processors. CM-tasks can either be basic modules that are treated as black-box codes or composed modules that can be decomposed into another set of CM-tasks. Hence, a hierarchical task structure arises.

The CM-task model distinguishes two different types of communication that are expressed by P-relations and C-relations:

1) **P-relation**
A precedence relation (P-relation) between CM-tasks A and B denotes that A produces output data required as an input for B. A data re-distribution might be necessary, when the executing processor groups of A and B are different or A produces the data in a different distribution than expected by B. Therefore, A must have finished and the data re-distribution needs to be carried out before B can start its execution. The P-relation is not symmetric and is represented by a directed edge.

2) **C-relation**
A communication relation (C-relation) between CM-tasks A and B results from communication between A and B during their execution. Therefore, to guarantee a correct execution A and B have to run in parallel on disjoint processor groups. The C-relation is symmetric and is represented by a bidirectional edge.

These relations can be captured by a CM-task graph $G = (V, E)$ with the set of nodes V corresponding to CM-tasks and the set of edges $E, E = E_P \cup E_C$, where E_P contains the directed edges introduced by P-relations and E_C contains bidirectional edges resulting from C-relations. Figure 1 shows an illustration of a CM-task graph. A CM-task application can be represented by a set of CM-task-graphs each corresponding to a composed module. The graph $G_C = (V, E_C)$ contains only the C-relations and $G_P = (V, E_P)$ contains only the P-relations. If there is a path from a node $u \in V$ to a node $v \in V$ in G_P then u and v have to executed one after another. Therefore, G_P has to be acyclic to allow a feasible execution order for the tasks. If there is a path from u to v in G_C then u and v have to be executed concurrently on disjoint subsets of the processors. As a consequence, there may not be both a path in G_P and a path in G_C between a pair of nodes u and v. If there is neither a path in G_P nor in G_C between u and v then both, a sequential and a concurrent execution of the corresponding CM-tasks are valid.

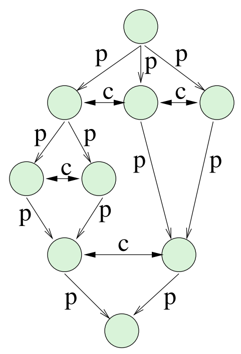

Figure 1. Example for a CM-task graph with P-relations and C-relations.

A CM-task schedule S assigns each CM-task $c \in V$ a processor group and a starting time. A feasible schedule has to follow all observations made above and furthermore has to guarantee that each processor executes one CM-task at most at any given point in time and that all input data are available before a CM-task starts its execution, i.e. all predecessor tasks in G_P have finished their execution and all necessary data re-distribution operations have been carried out.

3 Specification Language

The specification language describes the communication structure of a hierarchical CM-task application and consists of

- definitions of required data types and associated data distribution types;
- interface definitions of external CM-tasks (as basic modules) along with an estimation of the execution time;
- descriptions of internal CM-tasks (as composed modules);
- a main module as a starting point for the execution of the CM-task program.

Data types can be multi-dimensional arrays built up from basic types (integer, double). These data types are adequate for most regular scientific applications. Each data type is associated with a list of possible data distribution types describing how the elements of the data types are distributed across the available processors. Possible distribution types range from a replicated storage on all processors to block-cyclic distributions on arbitrary multi-dimensional proces-

Listing 1. Specification for one time-step of an IRK method.

```
const s=3;
cmmain irk_time_step (x,h:double:inout,
  it_vector, old_it_vector:vector:inout:replic)
{
  % local variables
  var vecs, oldvecs : vector[s];
  var vecxchg : vector;

  parfor (i = 0:s-1) {
    stage_vector (i, s, m, x, h, it_vector,
      vecs, oldvecs)[vecxchg];
  }

  compute_approx (h, it_vector, vecs);
  step_control (x, h, it_vector, old_it_vector,
    oldvecs, vecs);
}
```

sor grids.

The interface definition of a basic module includes two parameter lists; an input/output list for parameters that can be used to communicate along the P-relations; and a communication list for parameters to communicate along the C-relations. For each parameter, a data type and data distribution type is provided; the input/output list additionally includes an access type (in, out, inout). The estimation of the execution time is given as a formula depending on the number of executing processors p.

The description of composed modules includes a parameter list for input/output parameters similar to the basic modules, a list of local variables to store intermediate results and a structural description of the internal computations similar to imperative programming languages. The structural description consists of activations of CM-tasks (basic or composed modules) and constructs for the conditional execution of CM-tasks (*if*-statement), the repeated execution with dependencies between loop iterations (*while*-loop, *for*-loop) and without dependencies between loop iterations (*parfor*-loop). These constructs may be nested within each other. The P-relations and C-relations forming a CM-task graph are defined implicitly using the parameter lists of the activated CM-tasks.

Example

Examples for CM-task applications are solution methods for ordinary differential equations (ODEs). In particular, we consider iterated Runge-Kutta (IRK) methods which are explicit solution methods for initial value problems of non-stiff ODEs that offer potential for a parallel execution[17, 13]. In each time step, s stage vectors are computed by using m fixed point iteration steps. The values of s and m are determined by the employed RK method and known at compile time. At the end of each time step, the computed stage vectors are combined to form the new approximation vector and the step size for the next time step is computed.

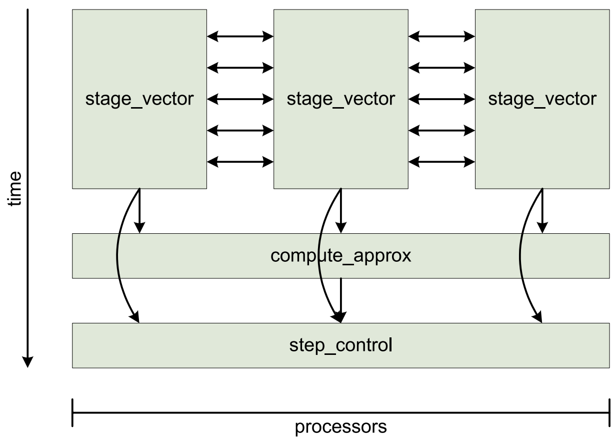

Figure 2. Task graph for IRK with $s = 3$ stage vectors and $m = 4$ fixed point iteration steps in the CM-task programming model. Each orthogonal communication is illustrated by a separate edge.

A parallel execution of an IRK method can make use of s processor groups each computing a single stage vector using a block-wise distribution of the elements. The computations of the stage vectors are not completely independent from each other; the execution of a fixed point iteration depends on all s stage vectors. Therefore, a data exchange between all participating processor groups is required before each fixed point iteration step. This data exchange can be realized by orthogonal communication meaning that only processors with the same rank in their group communicate with each other. This requires all s groups to contain the same number of processors.

Using the CM-task programming model the computation of one stage vector can be implemented as a CM-task `stage_vector` and the orthogonal communication can be modelled as C-relations. Figure 2 shows an illustration of the emerging task graph for one time step where each bidirectional edge represents an orthogonal communication step. The modelling of orthogonal communication is not possible using previous programming models based on M-tasks; the computation of the stage vectors has to be split into m tasks each computing one fixed point iteration. Figure 3 shows a possible task graph in such a programming model.

The resulting CM-task specification program for one time step is shown in Listing 1. The input of the time step is the current time `x`, and the current step size `h`. The current and previous approximation vectors (`it_vector` and `old_it_vector`) supply input data from the previous time step are updated in the current time step and are pro-

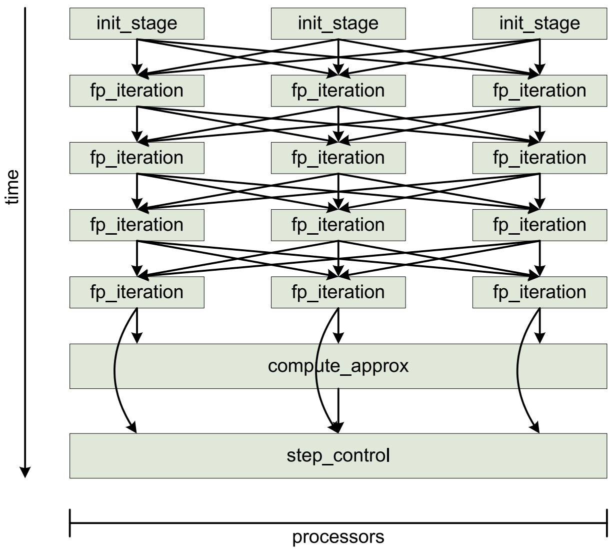

Figure 3. Task graph for IRK with $s = 3$ stage vectors and $m = 4$ fixed point iteration steps in the M-task programming model.

vided for the next time step, and are therefore declared as `inout` parameters. The data type `vector` and the data distribution type `replic` representing a replicated storage on all available processors have been declared previously. Local variables are required to store the computed stage vectors (`vecs`, `oldvecs`) and to perform orthogonal communication (`vecxchg`). The parallel loop (**parfor**) creates s independent instances of the CM-task `stage_vector`. The CM-task `compute_approx` determines the new approximation vector based on the computed stage vectors and the CM-task `step_control` calculates the new step size based on the current and previous approximation vectors and the values of the stage vectors computed in the last two fixed point iteration steps.

4 Transformation Framework

To support the development of CM-task programs, a compiler framework is provided that transforms CM-task programs given in the non-executable platform-independent specification language from Section 3 into executable parallel MPI programs. The framework integrates scheduling and load balancing methods, data distribution methods, as well as a generation process for the final MPI program. For the generation of parallel programs two approaches are supported. The static approach generates a parallel program at compile-time with a fixed schedule and all required data redistribution operations. It is especially suited for dedicated homogeneous platforms and requires detailed knowledge about the target platform (number of processors, communication and computational performance) and about a specific problem instance, e.g. the input data size. The semi-dynamic approach aims at an execution on a non-dedicated heterogeneous platform. A parallel program with an initial schedule is generated and a load balancing library adapts the schedule to the dynamic execution progress. A data redistribution library ensures a correct data placement at runtime.

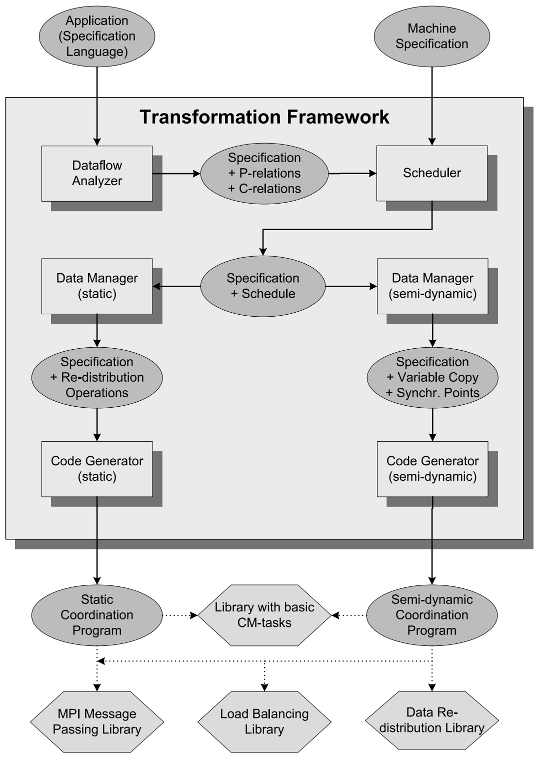
Application (Specification Language)
Machine Specification
Transformation Framework
Dataflow Analyzer
Specification + P-relations + C-relations
Scheduler
Data Manager (static)
Specification + Schedule
Data Manager (semi-dynamic)
Specification + Re-distribution Operations
Specification + Variable Copy + Synchr. Points
Code Generator (static)
Code Generator (semi-dynamic)
Static Coordination Program
Library with basic CM-tasks
Semi-dynamic Coordination Program
MPI Message Passing Library
Load Balancing Library
Data Re-distribution Library

Figure 4. Overview of the transformation framework.

Figure 4 gives an overview of the framework. Each of the transformation steps generates new information and outputs it as an augmented specification program. Support tools are provided that visualize the specification program and help the programmer to change the specification, e.g. to modify decisions of the framework. Each transformation step includes a parser that builds up the required symbol tables and internal structures from the specification. Currently, the transformation framework produces coordination code in the *C* programming language but can be extended easily to support other target languages.

Dataflow Analyzer The first transformation step analyzes the dependencies between the statements of a composed CM-task arising from using the same variables and inserts corresponding P-relations and C-relations into the specification. First, a *syntax tree* describing the hierarchical structure is created for each composed CM-task. The root node of the tree corresponds to the composed module itself and the leaf nodes (**call** nodes) are activations of other CM-

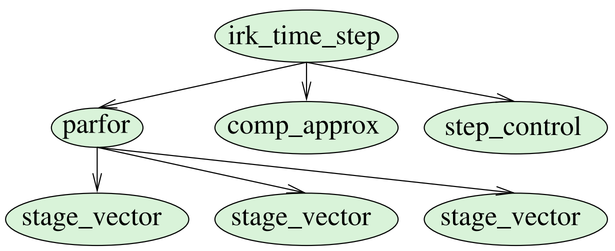

Figure 5. Syntax tree for the IRK method.

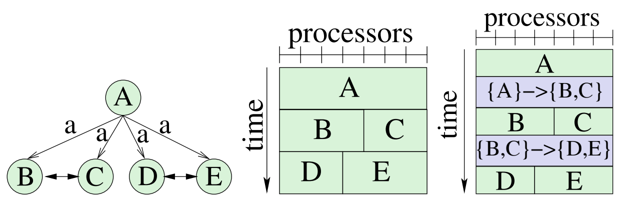

Figure 6. Example for the insertion of data re-distribution operations for a synthetic CM-task graph(left) with a valid schedule(middle) resulting in an augmented schedule(right).

tasks. The inner nodes correspond to control constructs of the specification language, i.e. **for**, **while**, **parfor** and **if**. The subtree below a **parfor** node is replicated according to the number of loop iterations. The **if** node comprises two distinct subtrees for the if-branch and for the else-branch. The children of a node are ordered according to the ordering in the specification program, i.e. the left sibling of a syntax tree node refers to the preceding statement in the specification program. This order plays a role when determining the data dependencies. Figure 5 shows a syntax tree for the IRK-method using $s = 3$ stage vectors.

Secondly, the P-relations are determined. In the syntax tree, P-relations can connect two sibling nodes or a parent-child pair. The insertion of the P-relations is based on the input and output parameters of the child nodes of a subtree. For each inner node representing a subtree, the output parameter set consists of all variables used by any child node as output parameter. The input parameter set contains all variables that are used as input parameter by any child node within the corresponding subtree for which there is no preceding write access to the variable within the same subtree.

Based on the input and output parameters, the P-relations are inserted by a top-down run over the syntax tree, capturing the dependencies that occur because different CM-tasks use the same variable as input or output parameter. If such a dependence occurs, the corresponding CM-tasks have to be executed in a specific order to ensure the correct computation. If a CM-task A writes a variable and another CM-task B later reads the same variable, it may be necessary to perform a re-distribution operation for that variable between the execution of A and B. Such a re-distribution is necessary if A and B use a different distribution type for the common variable or if different processor groups are used for the execution of A and B. The re-distribution operations are inserted by the static data manager.

Finally, the C-relations are computed by another top-down run on the syntax tree. C-relations connect two or more sibling nodes that are all **call** nodes which access a common communication parameter (defined in a second parameter list in square brackets). The C-relations are annotated at the appropriate inner nodes. Each C-relation is a list of the connected nodes.

Scheduler The scheduling itself is carried out on CM-task graphs which are built up from the syntax tree and the P-relations and C-relations. The schedule of a CM-task graph assigns each node an executing processor group, a starting time and a finishing time. To determine a schedule, the C-relations are eliminated from the graph by grouping all nodes that are connected by a C-relation into clusters and replacing each cluster with a single node. The resulting task graph can be scheduled using M-task scheduling techniques[5]. The created scheduling information is stored at the corresponding nodes of the syntax tree.

The scheduling is based on cost information. The costs for basic CM-tasks are supplied by the user together with the specification. The costs for composed CM-tasks are calculated according to the task structure. For the **for** and **while** nodes the costs are computed as the product of executed loop iterations and the execution time of the schedule for the corresponding subtree. The costs for an **if** node are the maximum of the execution times of the schedules generated for the if-branch and for the else-branch. The **parfor** node costs are equal to the execution time of the schedule of the corresponding subtree. Data re-distribution costs for P-relations are computed by determining the size of transmitted data and using a platform dependent startup time and byte transfer time that are defined in a separate machine specification file. This file additionally defines the number of processors of the target platform.

The **Static Data Manager** inserts data re-distribution nodes into the syntax tree of a CM-task application. This transformation step ensures a dataflow with correct data distributions for the final CM-task program. In general, a single re-distribution operation may have `N` disjoint source processor groups and `M` disjoint target processor group. This is illustrated for the example CM-task graph in Figure 6(left). In this example, the CM-task A produces output data for the independent CM-task B, C, D and E. For the valid schedule shown in Figure 6(middle), two re-distributions operations are required: one from CM-task A to CM-tasks B and C and one from CM-tasks B and C to CM-tasks D and E. The example with data re-distribution operations is shown in Figure 6(right). This example also demonstrates that the sources of the data re-distribution operations may differ from the sources of the P-relations.

In the first step, the static data manager determines the data distribution types at the inner nodes of the syntax tree by a bottom up run over the syntax tree. For each input/output parameter, the data distribution type of that child node is used which is computed first/last according to the scheduling information. For parameters that are in both, the input and the output parameter set, two different distributions may result.

The second step comprises the computation of the re-distribution operations for the P-relations. Each inserted data re-distribution node is annotated with the variable name, the source and target CM-tasks and a point in time where the operation should be performed. The target CM-tasks are equal to the target CM-tasks of the P-relations. The source CM-tasks and the execution point in time is determined using the scheduling information.

Static Code Generator The final transformation step compiles the specification program with all previously generated annotations into an executable coordination program which uses the MPI message passing library and a user supplied library that contains implementations of the basic CM-tasks. The coordination program consists of an initialization function that sets up all required MPI communicators, a coordination function for each composed CM-task and a finalization function that disposes the created communicators.

The code for a coordination function is constructed using the annotated syntax tree and a depth-first approach. For a given inner node the children are sorted with respect to their starting time according to the scheduling information; children with a lower starting time come first and in case of equal starting times the child with the lower processor numbers is considered first. The children are visited according to this list and depending on the type of the node one of the following code fragments are inserted:

- When entering the root node, the function header, the local variable declarations and memory allocation code is inserted. When leaving the root node, code for freeing the allocated memory is created. For the main composed module, additional code to execute the initialization and finalization function is inserted.
- For **if**, **while** and **for** nodes, the corresponding control construct in the target programming language is created.
- For **parfor** nodes, no action is performed because the expressed parallelism is already included in the scheduling information.
- For **call** nodes, a function call with the parameters according to the specification is generated. This function call is enclosed in a condition that ensures that it is executed by the processors according to the schedule.
- For **data re-distribution** nodes, either a re-distribution library function is called or the framework generates a re-distribution function that is inserted into the coordination program.

Semi-dynamic Data Manager In the semi-dynamic approach, the processor group layout is allowed to change at runtime possibly leading to complex changes of the required re-distribution operations. Therefore, the framework decides at compile-time, which variable accesses are performed to the original variable and which accesses are made to an auxiliary variable that has to be provided through copying the original variable. Write accesses are always performed to the original variable ensuring that it always contains the most recent values. The original variable can be accessed by one CM-task at most at any given point in time. Independent CM-tasks that may be executed in parallel always use different auxiliary variables. This approach ensures that only the re-distribution operations of a single CM-task A need to be considered if the processor group of A is changed at runtime.

Additionally, this transformation step marks the positions for performing the load balancing. These positions are points in time where all processors are available to perform a global restructuring of the processor groups and within sequential loops to use the executing results of previous loop iterations to adapt to the execution behavior. The user can add additional positions or delete positions suggested by the framework.

Semi-dynamic Code Generator The semi-dynamic code generator creates a coordination program that uses the MPI message passing library, a load balancing library and a data re-distribution library. The coordination program includes a coordination function for each defined CM-task-graph that creates the required processor groups and executes the CM-tasks with the according communicators. The code generation for a composed CM-task is performed similar to the static code generator.

The load balancing library is initialized at program start with the CM-task-graph representing the whole application. The library may influence the execution of the CM-task program at the points in time specified by the previous transformation step. Therefore, calls to the library are inserted which provide information about the current execution status of the application, the current processor groups, the current execution order of the CM-tasks and measured runtimes of previous executions of CM-tasks. If the library detects a suboptimal execution of the application, e.g. load imbalances, it may output new processor groups and a new execution order for the CM-tasks.

The data re-distribution library is invoked each time a CM-task finishes its execution. It determines and carries out all necessary copy and re-distribution operations for the current processor group layout.

5 Runtime Results

Benchmark tests were executed to evaluate the advantages of an orthogonal implementation based on CM-tasks over a task parallel version based on M-tasks and a classical data parallel computation scheme for the IRK methods used as an example application in Subsection 3. The presented runtime results are obtained for ODE systems that result from a spatial discretization of the 2D Brusselator equation [7]. The resulting ODE systems are sparse: each component of the right-hand side function **f** of the ODE system has a fixed evaluation time that is independent of the size of the ODE system; thus, the evaluation time for the entire function **f** increases linearly with the size of the ODE system. The presented figures show the execution time of one time step, obtained by dividing the total execution time by the number of time steps performed. A typical integration may consist of tens of thousands of time steps, thus leading to a large overall execution time.

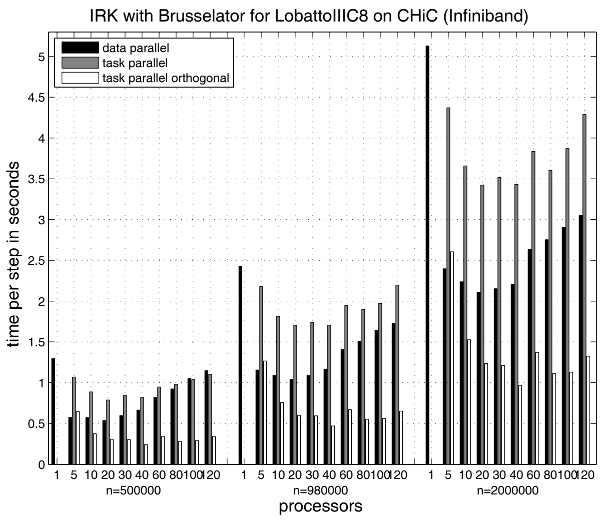

Figure 7. Runtimes of the IRK method with $s = 5$ stage vectors for Brusselator on CHiC with Infiniband network.

Figure 7 shows the execution times of one time step of the IRK method on the CHiC cluster. This cluster consists of 538 nodes each equipped with two Opteron 2218 processors with a clock rate of 2.6 GHz. The nodes are interconnected by a 10 GBit/s infiniband network and MVAPICH2 was used as an MPI library. As basic RK method, the LobattoIIIC8 method with $s = 5$ stage vectors and $m = 7$ fixed point iteration steps was used. The results show that the orthogonal program version is considerably faster than both, the data parallel and the standard task parallel version for a higher number of processors. A data parallel execution achieves better results only for $p = 5$ processors, because the overhead of the communication operations is still small and the additional re-distribution operations required for a task parallel execution can be avoided.

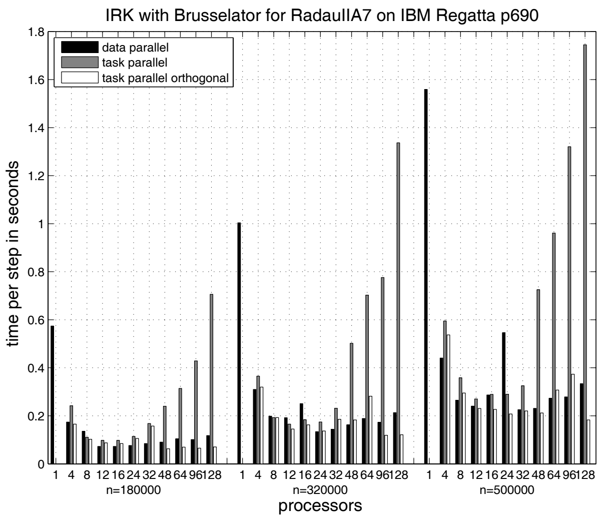

Figure 8. Runtimes of the IRK method with $s = 4$ stage vectors on IBM Regatta.

The runtimes for a Regatta p690 system are shown in Figure 8 for the RadauIIA7 method which uses $s = 4$ stage vectors and $m = 6$ fixed point iteration steps. The system consists of 41 nodes with 32 IBM Power 4+ processors clocked at 1.7 GHz each connected by a high performance switch with a bandwidth of 1400 MB/s. The runtimes for the standard M-task version are not competitive due to an expensive two-step communication after each fixed point iteration step which includes a broadcast within each processor group followed by an exchange between all processor groups. An orthogonal execution is faster compared to a pure data parallel execution for most test runs even though the large SMP nodes offer a high communication performance. Therefore, these results show that the CM-task programming model is also suitable for shared memory platforms.

6 Related Work

The programming model presented in this paper extends approaches based on mixed task and data parallelism by capturing additional communication patterns. The benefits of combining task and data parallelism are discussed in [3] and [2, 16] gives an overview of support systems and models. Language extensions are often based on the

HPF data parallel language, e.g. BCL[4] and see [6] for an overview. Spar[15] is a set of language extensions for Java. The paradigm compiler[9, 12] is a parallelizing compiler for mixed task and data parallelism with an integrated scheduler. A framework that allows the derivation and scheduling of task graphs from sequential programs is presented in [19]. The Lithium environment[1] includes task and data parallel skeletons that may be nested within each other. The ASSIST system[18] supports the hierarchical composition of parallel and sequential modules as generic graphs. In contrast to our model a communication between parallel modules during their execution is not possible.

A lot of research has been invested in the development of the BSP (bulk synchronous parallelism) model and there exists a programming library (Oxford BSP library) that allows the formulation of BSP programs in an SPMD style [8, 11]. NestStep extends the BSP model by supporting group-oriented parallelism by nesting of supersteps and a hierarchical processor group concept [10]. NestStep is defined as a set of extensions to existing programming languages like C or Java and is designed for a distributed address space.

7 Conclusions

In this paper, we have presented a new parallel programming model which extends existing programming models that are based on a combination of task and data parallelism. Similar to existing models, the new model allows the decomposition of an application into a set of cooperating parallel tasks that can be executed on a flexible number of processors. In addition, the extended model supports a new type of communication that enables the exchange of data between running tasks and is therefore able to express more complex communication patterns, e.g. orthogonal communication.

For the implementation of the programs in the extended model, we have introduced a specification language that can be compiled by a transformation framework into an executable program. The framework includes multiple transformation steps where the user can observe each step and adjust the intermediate results. The derived target program can be either statically compiled for a fixed number of processors on a homogeneous target platform or use a load balancing library to semi-dynamically adapt to a flexible number of processors or platform heterogeneity.

To demonstrate the benefits of the new programming model, orthogonal implementations of solvers for ordinary differential equations were considered. In particular, the benchmark results obtained for the iterated Runge-Kutta methods on different target platforms show a significant performance improvement compared to data parallel or pure task parallel realizations.

References

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

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

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

[4] M. Diaz, B. Rubio, E. Soler, and J. Troya. A border-based coordination language for integrating task and data parallelism. *J. Par. Distrib. Comput.*, 62(4):715–740, 2002.

[5] J. Dümmler, R. Kunis, and G. Rünger. A Scheduling Toolkit for Multiprocessortask Programming with Dependencies. In *Euro-Par 2007, Parallel Processing*, volume 4641 of *LNCS*, pages 23–32, Rennes, France, 2007. Springer.

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

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

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

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

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

[11] W. McColl. Universal Computing. In *Proceedings of the EuroPar'96*, Springer LNCS 1123, pages 25–36, 1996.

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

[13] T. Rauber and G. Rünger. Parallel Execution of Embedded and Iterated Runge–Kutta Methods. *Concurrency: Practice and Experience*, 11(7):367–385, 1999.

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

[15] H. Sips and K. van Reeuwijk. An integrated annotation and compilation framework for task and data parallel programming in java. In *Proc. of 12th Int. Conf. on Par. Comp. (ParCo'03)*, 2004.

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

[17] P. J. van der Houwen and B. P. Sommeijer. Iterated runge-kutta methods on parallel computers. *SIAM J. Sci. Stat. Comput.*, 12(5):1000–1028, 1991.

[18] M. Vanneschi. The programming model of assist, an environment for parallel and distributed portable applications. *Parallel Comput.*, 28(12):1709–1732, 2002.

[19] W. Zimmermann and W. Löwe. Foundations for the integration of scheduling techniques into compilers for parallel languages. *Int. J. Comp. Science & Engineering*, 1(3/4), 2005.