Combined Scheduling and Mapping for Scalable Computing with Parallel Tasks

Jörg Dümmler[1], Thomas Rauber[2], and Gudula Rünger[3]

To cite this version:

Dümmler, J.; Rauber, T.; Rünger, G.: Combined Scheduling and Mapping for Scalable Computing with Parallel Tasks. In: Scientific Programming, Bd. 20, Nr. 1: S. 45-67. IOS Press – ISSN 1058-9244, 2012. DOI: 10.3233/SPR-2012-0338

[1] **Corresponding Author**, Chemnitz University of Technology, Department of Computer Science, Str. der Nationen 62, 09111 Chemnitz, Germany
Phone: +49 (0) 371 531 31494, Fax: +49 (0) 371 531 831494,
Email: djo@cs.tu-chemnitz.de

[2] Bayreuth University, Angewandte Informatik II,
95440 Bayreuth, Germany, Email: rauber@uni-bayreuth.de

[3] Chemnitz University of Technology, Department of Computer Science,
09111 Chemnitz, Germany, Email: ruenger@cs.tu-chemnitz.de

Abstract

Recent and future parallel clusters and supercomputers use symmetric multiprocessors (SMPs) and multi-core processors as basic nodes, providing a huge amount of parallel resources. These systems often have hierarchically structured interconnection networks combining computing resources at different levels, starting with the interconnect within multi-core processors up to the interconnection network combining nodes of the cluster or supercomputer. The challenge for the programmer is that these computing resources should be utilized efficiently by exploiting the available degree of parallelism of the application program and by structuring the application in a way which is sensitive to the heterogeneous interconnect.

In this article, we pursue a parallel programming method using parallel tasks to structure parallel implementations. A parallel task can be executed by multiple processors or cores and, for each activation of a parallel task, the actual number of executing cores can be adapted to the specific execution situation. In particular, we propose a new combined scheduling and mapping technique for parallel tasks with dependencies that takes the hierarchical structure of modern multi-core clusters into account. An experimental evaluation shows that the presented programming approach can lead to a

significantly higher performance compared to standard data parallel implementations.

Keywords: Scheduling, Algorithms, Performance Measurements, Parallel Tasks, M-tasks, Mapping, Multi-core, Scalability

1 Introduction

Recent and future parallel clusters and supercomputers offer a very large number of parallel processing units. The immense increase in parallelism of these architectures is caused by the use of multi-core and many-core processors. Today most parallel machines are equipped with processors comprising two to eight cores; within a few years a single processor is expected to provide tens or even hundreds of execution cores. For the application programmer, the architectural development towards multi-core systems poses the challenge to provide and implement application codes with a suitable degree of potential parallelism. The degree of parallelism within a parallel application code depends on the characteristics of the problem to be solved but also on the parallel programming model used for designing and coding the parallel application. In this article, we propose to use hierarchically structured parallel tasks for developing application programs for such large parallel systems. Each parallel task can be executed by an arbitrary number of cores or processors; therefore, they are also called multiprocessor tasks (M-tasks).

The M-task programming model can be used to structure parallel programs in a flexible way by expressing the available degree of task parallelism in form of M-tasks. This can, for example, be used to combine the benefits of task and data parallelism by using data parallelism within M-tasks and by expressing task parallelism as interactions between M-tasks. An M-task program is subdivided into a set of M-tasks, each working on a different part of the application. A coordination structure (e.g. a directed acyclic graph) describes how the M-tasks cooperate with each other and which dependencies have to be considered for a correct parallel execution. The coordination structure also identifies sets of M-tasks that might be executed concurrently with respect to each other because there are no dependencies. The M-task coordination structure can be organized hierarchically. The subdivision stops with basic M-tasks that are not further subdivided. The basic M-tasks can be implemented using an SPMD (single program, multiple data) programming style, e.g., by employing MPI or OpenMP, and can run on an arbitrary number of processors or cores.

The coordination structure can be defined by an application-specific specification that is independent of the hardware and the interconnection network of the target platform. This decouples the specification of parallelism from the actual parallel execution on a specific parallel platform. Thus, the parallel execution can be modified without changing the specification of parallelism, leading to a portability of efficiency. For a specific target platform, the M-tasks can be executed such that the computational work is balanced and the resulting communication overhead is at a minimum. The advantage of this approach is that the available degree of parallelism can be increased by defining a suitable M-task structure and that

communication within M-tasks can be restricted to subsets of the available processors or cores. Thus, the communication overhead of the application can be reduced especially on distributed memory platforms leading to an improved scalability.

An M-task application program and its coordination structure offer several possibilities for a parallel execution, differing in the order in which the M-tasks of a program are executed and in the assignment of subsets of processors or cores to the individual M-tasks for execution. These decisions are made in separate scheduling and mapping steps, which use the coordination structure as input. On different parallel architectures, different versions of the M-task program might lead to the most efficient and scalable execution. To find an optimal M-task program version is an NP-hard problem, which is usually solved by scheduling heuristics or approximation algorithms [28]. Most of those existing M-task scheduling algorithms are defined for homogeneous systems or distributed grid-like systems.

In previous work [18], we have developed a combined scheduling and mapping algorithm for the execution of M-task applications on heterogeneous multi-core platforms. In this article, we extend this approach to a broader class of applications and also consider hybrid MPI+OpenMP execution schemes. In particular, the contribution of this article includes:

- to propose the M-task programming model as a suitable programming model for large multi-core systems, which can increase the potential parallelism due to a mixture of task and data parallelism;
- to suggest a combined scheduling and mapping strategy for M-task programs, which extends existing scheduling approaches for the use in hierarchical multi-core systems;
- to investigate the performance impacts of different mapping strategies for several benchmarks including solvers for ordinary differential equation (ODEs) on multi-core SMP systems. One-step ODE solvers are important in scientific computing but have by their nature a limited degree of parallelism; thus it is important to provide suitable parallel implementations [9]. We also consider solvers for partial differential equations (PDEs) from the NAS parallel benchmark suite. The investigations for dual-core and quad-core systems show that the performance of these applications is significantly impacted by both, the scheduling decision and the mapping strategy applied.

The article is organized as follows. Section 2 gives a short description of the M-task programming model and discusses the implementation of M-task programs. Section 3 describes scheduling algorithms and mapping strategies for M-tasks. Section 4 presents a detailed experimental evaluation of the mapping strategies

for different parallel systems. Section 5 discusses related work and Section 6 concludes the article.

2 M-Task Programming

This section describes the M-tasks programming model and discusses implementation approaches for designing M-task programs.

2.1 General Approach

The M-task programming model is a programming style to code parallel programs in a mixed task and data parallel way using a set of cooperating parallel tasks (M-tasks). An M-task is a piece of parallel program code (e.g. implemented as a function in *C* or a subroutine in *FORTRAN*) that operates on a specific set of input parameters and produces a specific set of output parameters. Internally, M-tasks are implemented such that an execution on an arbitrary number of processors or cores is possible. Therefore, there may be internal communication operations to exchange data between the processors or cores executing the same M-task.

An M-task can be basic or composed. A basic M-task is implemented by the application developer in an SPMD programming style, e.g., using MPI, OpenMP or a hybrid MPI+OpenMP approach. Composed M-tasks include activations of other basic or composed M-tasks. Thus, M-task programs can be structured hierarchically.

There may exist input-output relations between the M-tasks that have to be taken into consideration for the coordination of the M-tasks. An input-output relation exists between an M-task M_1 and an M-task M_2 if M_1 produces output data that is required as an input for M_2. These relations might lead to the necessity of data re-distribution operations if M_1 and M_2 are executed on different sets of cores or if M_1 produces its output in a different data distribution than expected by M_2. The data distribution of an input or output parameter of an M-task M defines how the elements of this data structure are distributed over the set of cores executing M. An example is the cyclic distribution of the elements of a one-dimensional array.

The structure of an M-task program can be represented by an M-task graph $G_M = (V, E)$ which is a directed acyclic graph whose nodes represent the M-tasks of the program. A directed edge $e \in E$ connects two M-tasks M_1 and M_2 if there is an input-output relation between M_1 and M_2. Examples for M-task graphs are the macro data-flow graphs used in the PARADIGM compiler [3] or the SP-graphs used in the TwoL model [43]. Figure 1 (left) shows an example for an M-task graph consisting of nine M-tasks.

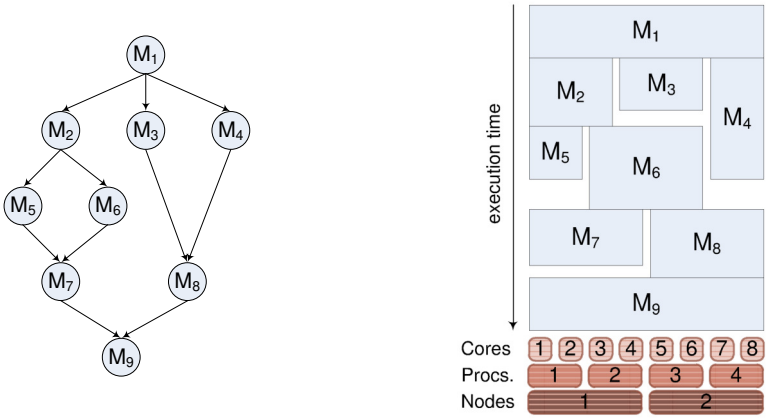

Fig. 1: (Left) Example for an M-task graph consisting of M-tasks $\{M_1, \ldots, M_9\}$ and directed edges denoting input-output relations between the M-tasks. (Right) Illustration of a suitable M-task schedule for the example M-task graph on a platform consisting of two nodes, each one equipped with two dual-core processors.

Typically, there exist many possible parallel execution orders for a given M-task graph differing in the scheduling and mapping of M-tasks to subsets of cores. If the M-tasks M_1 and M_2 are connected by an edge in the M-task graph the execution of M_1 must have been finished and all data re-distribution operations required between M_1 and M_2 must have been carried out before the execution of M_2 can be started. For independent M-tasks, i.e., M-tasks that are not connected by a path in the M-task graph, a concurrent execution on disjoint subsets (groups) of the available cores as well as an execution one after another are possible. Figure 1 (right) shows an illustration of a schedule for the example M-task graph.

For the parallel execution of an M-task program on a hierarchical multi-core platform there exist several execution schemes differing in

a) the number of cores assigned to each M-task;
b) the execution order for independent M-tasks (concurrently or one after another or a mixed order);
c) the assignment of specific processors (or cores) of the execution platform to specific groups of cores executing M-tasks (mapping).

Different execution schemes lead to different communication patterns between the processes of an M-task application and, thus, lead to different execution times. In particular, different communication times may result for the communication within the M-tasks as well as for data re-distribution operations that might be required

between cooperating M-tasks to adjust the distribution of common data structures. An efficient execution scheme can be selected by suitable mapping and scheduling strategies as discussed in the next section.

2.2 Implementation of M-task programs

The decomposition of a parallel application into M-tasks is usually part of the application design process, since the restructuring of existing monolithic applications might be expensive. The coding of an M-task program requires the implementation of the M-tasks and the specification of the cooperation between the M-tasks of the parallel program. Also, the scheduling decision can be part of the program code. There are different ways for capturing these implementation details, including language extensions, library-based approaches and coordination-based approaches. In the following, we discuss some important issues concerning the specification, scheduling and execution of M-task programs.

2.2.1 Specification of M-task programs

Depending on the programming approach, the coordination structure of the M-task program, i.e., the M-task graph, can be *implicit* or *explicit*. An implicit coordination structure exists when the entire program is coded by hand, e.g., using MPI. In this case, the programmer is responsible for the construction of the groups of cores, the execution of the M-tasks on these groups, and the execution of the data re-distribution operations.

For an explicit coordination structure, a suitable programming tool is required that allows the formulation of interactions between M-tasks according to a graph structure as depicted in Fig. 2. One possibility is to use a coordination language with operators expressing dependencies (or independencies) between M-tasks of a parallel program. The dependencies can be given by the names of the variables causing the input-output relations. Such a tool is the TwoL compiler [43] or its successor, the CM-task compiler [17], in which the coordination structure can be expressed using appropriate coordination constructs. The resulting specification program is not executable, but is translated into a final C with MPI program. A translation into other parallel languages is also possible. Other coordination-based approaches include the network of Tasks model [36], the performance-aware composition framework [24] as well as the Paradigm compiler [40]. These approaches provide the advantage of decoupling the specification from the execution on a specific parallel platform. Thus, the programmer is relieved to consider any platform-specific details such as the determination of a suitable schedule. Moreover, the communication between M-tasks, i.e. the data re-distribution operations resulting

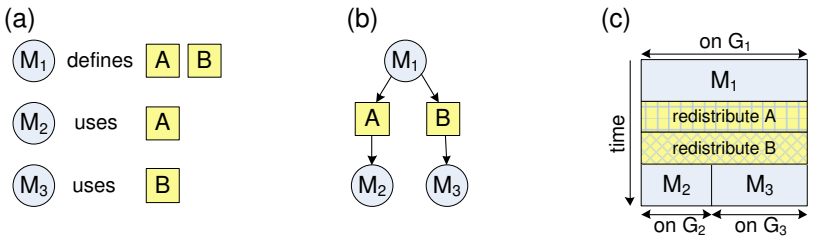

Fig. 2: Illustration of the programming with M-tasks: (a) the M-tasks M_1, M_2, M_3 define or use data A or B; (b) the coordination structure describes the relations between the M-tasks: M_2 uses data A defined by M_1 and M_3 uses data B defined by M_1; (c) scheduling of M_1, M_2, and M_3 with re-distribution for data A and B and a concurrent execution of M_2 and M_3 on subgroups G_2 and G_3, respectively.

from the input-output relations, can be inserted by the compiler tool resulting in a reduced implementation effort compared to an implicit coordination structure. In this article, we assume an explicit coordination structure that is defined using a specification program, see Sect. 2.2.3 for an example.

2.2.2 Scheduling

The translation of an M-task specification into the final executable program depends on the scheduling decision. A scheduling of the M-tasks of an M-task program determines a relative order of the execution of the M-tasks. Since the M-tasks are parallel program codes, this includes the assignment of a group of cores to M-tasks. Constraints of the scheduling are the input-output relations between different M-tasks as well as the total number of cores assigned to the entire parallel program: M-tasks with input-output relations have to be executed one after another, and M-tasks executed in parallel are assigned to disjoint subsets of the available set of cores, such that the total size cannot exceed the number of cores available.

In principle, the scheduling can be planned statically or dynamically during program execution. For a static scheduling, the M-task graph must be known before program start. The static case has the advantage that a sophisticated scheduling algorithm can be applied to the complete M-task graph. The static scheduling can be computed by hand and the resulting schedule can be included directly in the application program. This approach is sometimes useful for M-task programs that are solely written in MPI. A static scheduling can also be determined by a compiler tool. In this case, it is beneficial to make the coordination of the M-tasks explicit, e.g., in form of an M-task graph, so that the scheduling decision can be expressed in form of a suitable coordination program before the translation into the final parallel

program.

For a dynamic scheduling, subsets of cores are assigned to M-tasks at runtime, depending on the availability of free cores. This approach can also handle the dynamic or recursive creation of M-tasks, which is suitable for adaptive computations or divide-and-conquer algorithms. The Tlib library supports such applications [44]. In this article, we consider the static case.

2.2.3 Example

As an example for M-task programs, we consider the numerical solution of initial value problem (IVPs) of systems of ordinary differential equations (ODEs), which are problems of the form:

$$\mathbf{y}'(t) = \mathbf{f}(t, \mathbf{y}(t)), \quad \mathbf{y}(t_0) = \mathbf{y}_0$$

where $\mathbf{f} : \mathbb{R} \times \mathbb{R}^n \to \mathbb{R}^n$ is a given real-valued vector function (right-hand-side function), $\mathbf{y} : \mathbb{R} \to \mathbb{R}^n$ is the unknown solution function, and $\mathbf{y}_0$ is the given initial value of the solution function at time t_0.

Numerical solution methods for ODE IVPs perform a time-stepping procedure consisting of a potentially large number of discrete time steps $\kappa = 0, 1, 2, \ldots$ corresponding to time t_κ. Starting at time t_0 with the initial approximation $\boldsymbol{\eta}_0 = \mathbf{y}(t_0) = \mathbf{y}_0$, a new approximation $\eta_{\kappa+1}$ is computed at each time step κ using the previous approximation η_κ and, depending on the specific method, additional approximations computed previously. The procedure repeats until the end of the integration interval $[t_0, t_e]$ is reached. The local error is estimated at each time step and the step size $h_\kappa = t_{\kappa+1} - t_\kappa$ is adapted accordingly such that a specified accuracy is maintained.

In the following, we consider an extrapolation method, which is an explicit one-step solution method for systems of ODEs. One time step of this method computes R approximations using R different step sizes $h_1, \ldots, h_R$ and combines these approximations into a final approximation of higher order. The i-th approximation is computed by the consecutive execution of i micro steps using step size $h_\kappa/i, i = 1, \ldots, R$ for each micro step. The micro steps of the same approximation have to be computed one after another, but the micro steps of different approximations are independent of each other.

The extrapolation method can be implemented using three different basic M-tasks. The step size `h` and the time index `t` are initialized by M-task `init_step`. The M-task `step` computes a single micro step, and the M-task `combine` is responsible the determination of the final approximation vector of a time step and for the computation the time index and step size for the next time step.

```
1  const R = 4;          // number of approximations
   const Tend = ...;     // end of integration interval
3  cmmain EPOL (eta_k:vector:inout:replic) {
     // definition of local variables
5    var t, h : scalar;  // time and step size
     var V : Rvectors;   // approximation vectors
7    var i, j : int;
     // module expression
9    seq {
       init_step (t,h);
11     while (t < Tend) { // time stepping loop
         seq {
13         parfor (i = 1:R) {
             for (j = 1:i) {
15             step (j,i,t,h,eta_k,V[i]); } }
           combine (t,h,V,eta_k);
17 } } } }
```

Fig. 3: Specification program for the extrapolation method.

Figure 3 shows a specification program of the composed M-task `EPOL` for the CM-task compiler [17]. The module expression (lines 9-16) defines possible execution orders of the M-tasks. The operator `seq` specifies an execution one after another due to input-output relations. The operators `for` and `parfor` define loops with or without input-output relations between loop iterations, respectively. The declarations of the data types (`scalar`, `vector` and `Rvectors`), data distribution types (`replic`) and the interfaces of the basic M-tasks have been omitted in the figure. In general, the CM-task compiler supports multidimensional arrays as data types and arbitrary block-cyclic distributions over multidimensional processor meshes as data distribution types.

The CM-task compiler extracts two M-task graphs from the specification of the extrapolation method. In the first (upper level) M-task graph, the entire `while` loop (lines 11-16) is represented by a single node. The second (lower level) M-task graph represents the body of the `while` loop, i.e., one time step of the extrapolation method. Figure 4 shows an illustration of these graphs for $R = 4$ approximation vectors. Loop unrolling has been used for both, the `parfor` loop (line 13) and the `for` loop (line 14). The M-task graphs constructed include a unique *start* node and a unique *stop* node that are inserted automatically to mark the start or the end of the execution, respectively. The nodes representing M-task `step` include a pair (i, j) of numbers giving the iteration of the i (`parfor`) loop and j (`for`) loop, respectively.

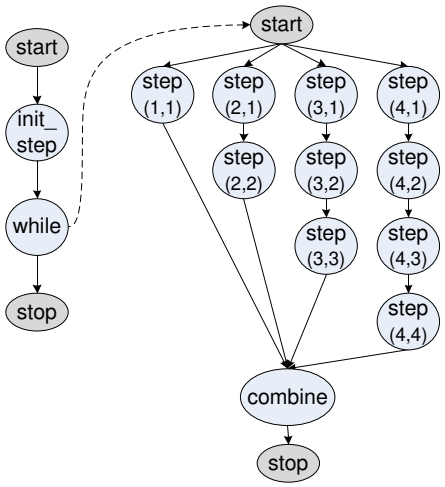

Fig. 4: Hierarchical M-task graph produced by the CM-task Compiler from the specification program from Fig. 3 for $R = 4$.

The M-task graphs are scheduled using a hierarchical approach, which means that the available processors or cores for scheduling the lower level M-task graph are determined by the processors or cores assigned to the `while` loop in the schedule of the upper level M-task graph. The advantage of this hierarchical approach is that the M-task graphs constructed are acyclic, since the information on the repeated execution of the loop body is encoded in the node representing the entire loop. In the following, we focus on the scheduling and mapping of a single non-hierarchical M-task graph.

3 Scheduling and Mapping

Executing an M-task program on a hierarchical multi-core machine requires several steps: scheduling the execution order of the M-tasks, determining the number of cores assigned to each M-task and mapping the M-tasks to specific cores. In the following, we assume that the individual M-tasks of the application are implemented for distributed memory, e.g., using a pure MPI or a hybrid MPI+OpenMP implementation. Thus, an M-task is assumed to be executable by processor cores located on different nodes of the parallel architecture. If multiple cores of the same node are assigned to a hybrid MPI+OpenMP M-task by the scheduling and mapping procedure, a single MPI process and an appropriate number of OpenMP

threads is started on this node.

3.1 Cost Model for M-tasks

The scheduling decision is based on the following cost model for the execution times of the M-tasks: The costs T of a single M-task M depend on the computational work of M, the number of cores used for the execution of M, and the mapping pattern describing the interconnection of the cores used for the execution. The costs of M for q cores is assumed to be

$$T(M, q, mp) = T_{comp}(M)/q + T_{comm}(M, q, mp)$$

where the computational work is captured by the sequential execution time T_{comp} (M) divided by the number of cores q. The total internal communication time $T_{comm}(M, q, mp)$ depends on the mapping pattern mp. For the computational part a linear speedup is assumed, ignoring performance effects resulting from the memory hierarchy or load imbalances between cores executing the same M-task. This assumption facilitates the derivation of a cost expression while still maintaining a good accuracy, see e.g. [25] for a comparison of predicted and measured execution times for the ODE solvers studied in this article. Other types of applications might require a more sophisticated performance model. This can be incorporated into the scheduling and mapping approach presented in this article by using an appropriate cost function.

For example, the costs for M-task `step`, which computes a micro step of the extrapolation method (see Subsect. 2.2.3) can be approximated by function

$$T(\texttt{step}, p, mp) = \frac{n(2t_{op} + t_{eval}(f))}{q} + T_{mb}(q, n, mp).$$

In this function, t_op denotes the average execution time of an arithmetic operation, $t_{eval}(f)$ is the time required to evaluate a single ODE from the ODE system f of size n, and $T_{mb}(q, n, mp)$ is the communication time of a multi-broadcast operation depending on the number of participating cores q, the size of transmitted data n and the mapping pattern mp.

Additional data re-distribution costs $T_{Re}(M_1, M_2, q_1, q_2, mp_1, mp_2)$ may also occur between cooperating M-tasks M_1 and M_2 where M_i is executed by q_i cores using mapping pattern mp_i, $i = 1, 2$. Re-distribution costs are determined by the size of data transmitted between M_1 and M_2, and the startup and byte-transfer time of the interconnection network.

3.2 Scheduling Algorithm

The scheduling algorithm determines the execution order for the M-tasks of a task graph. In particular, the scheduling algorithm decides whether independent M-tasks are executed concurrently to each other using disjoint groups of execution cores or whether a sequential execution is used, employing all available execution cores for each of the independent M-tasks one after another. The algorithm assumes that each M-task is executed within a specific execution time interval by the assigned cores. The distribution of the workload among these cores and the load balancing between these cores are not part of this algorithm.

In the following, we use a layer-based scheduling algorithm which partitions the M-task graph into layers of independent M-tasks and schedules the layers one after another [42]. The algorithm proceeds in three steps: In the first step, the algorithm identifies linear chains of M-tasks in the M-task graph $G = (V, E)$ and replaces each chain with a single node. A linear chain is a subgraph of G consisting of $n, n \geq 2$, nodes with the following properties:

- There is a unique entry node that precedes all other nodes of the linear chain.
- There is a unique exit node that succeeds all other nodes of the linear chain.
- All nodes of the chain except the entry node have exactly one predecessor that is also part of the linear chain.
- All nodes of the chain except the exit node have exactly one successor that is also part of the linear chain.

A linear chain is of maximum size if it is not possible to add an additional node to this chain without violating one of the conditions stated above. Each chain of maximum size is replaced by a single node; the costs are the accumulated computation and communication costs of the M-tasks included. This step can reduce the number of nodes in an M-task graph and also guarantees that the M-tasks of the same linear chain are scheduled onto the same set of cores, so that expensive data re-distribution operations between these M-tasks can often be avoided. For example, in the M-task graph for one time step of the extrapolation method from Subsect. 2.2.3 (see also Fig. 4), the micro steps of the same approximation form a linear chain, see Fig. 5 (left) for an illustration.

In the second step, the scheduling algorithm partitions the M-task graph into layers of independent M-tasks. For this purpose, a greedy algorithm runs over the M-task graph in a breadth-first manner and puts as many independent nodes as possible in the current layer. This approach is especially useful for applications that consist of multiple consecutive phases, each consisting of a set of independent M-tasks. An example are the solvers for systems of ODEs considered in Sect. 4.2. In general, the greedy approach constructs layers with as many M-tasks as possible and, thus, provides great flexibility for the scheduling decision. Figure 5 (right)

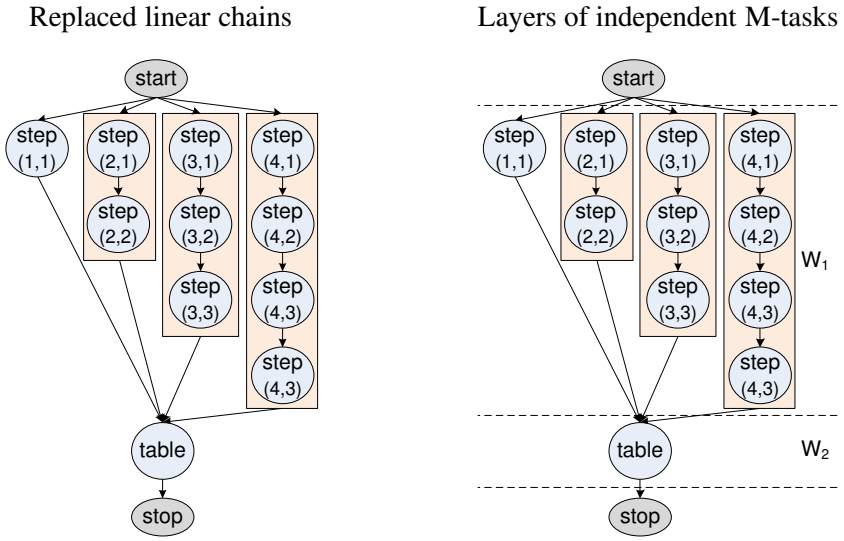

Fig. 5: (Left) Identification of the linear chains in the M-task graph for one time step of the extrapolation method. The nodes replacing the linear chains are depicted as rectangles. (Right) Partitioning of the M-task graph (after replacement of the linear chains) into two layers W_1 and W_2.

illustrates the decomposition of the M-task graph for one time step of the extrapolation method into layers. The *start* node and the *stop* node are not assigned to any layer, since these nodes do not carry computations.

In the third step, the layers of the M-task graph are scheduled one after another. Within a layer, the M-tasks can be scheduled in an arbitrary way. Especially, the set of execution cores can be partitioned into an arbitrary number g of subsets of cores where each subset is responsible for the execution of a subset of the M-tasks of the layer. The scheduling algorithm makes two assumptions: (a) the number of subsets and their size remains constant during the execution of the M-tasks of one layer, and (b) the subsets are built using symbolic cores, which are interconnected by a homogeneous network. Assumption (a) is reasonable, because a reorganization of the group structure during the execution of the M-tasks of one layer is usually quite expensive. Assumption (b) is an abstraction which allows the separation of scheduling and mapping; the separate mapping of the symbolic cores to the physical cores of a hierarchical architecture is described in Subsect. 3.4.

The total number of symbolic cores used is equal to the number of physical cores of the target architecture. In the following, the costs for an M-task M on a set of p symbolic cores are denoted as $T_{symb}(M, p)$. These costs are computed

by $T_{symb}(M,p) = T(M,p,dmp)$ using a default mapping pattern dmp, which denotes a mapping where the slowest interconnection network of the architecture is used for all communication operations executed within M. Thus, $T_{symb}(M,p)$ is an upper limit of the execution time of M on p cores of the target system.

Algorithm 1 shows pseudo-code of the scheduling procedure. For each layer W, the scheduling algorithm considers all numbers $g \in \{1,\ldots,P\}$ of subsets of symbolic cores where P is the total number of cores of the platform (line 5). The output of the algorithm is the number g_{min} of subsets that leads to the minimum execution time

$$T_{min} := \min_{1 \le g \le P} T_{act}(g)$$

where $T_{act}(g)$ is the total execution time of layer W using g subsets of cores. The time $T_{act}(g)$ is determined as follows: First, the set of symbolic cores is partitioned into g subsets $G = \{G_1,\ldots,G_g\}$ (line 6). The subsets have the same size, which may later be adapted in the adjustment step. The assignment of M-tasks to specific subsets is performed by a modified greedy linear-time scheduling algorithm for uniprocessor tasks without dependencies [46] with a proven sub-optimality bound of $4/3$. This sub-optimality bound does not hold for M-task layers, but the algorithm shows good results in practice. The M-tasks of the layer are considered one after another in decreasing order of their execution time and are assigned to the subset with the currently smallest accumulated execution time (line 10). The accumulated execution time of a subset G_l is defined as the sum of the parallel execution times of the M-tasks previously assigned to G_l. After all M-tasks are assigned, the subset with the maximum accumulated execution time determines the total execution time $T_{act}(g)$ of the entire layer using g subsets of cores (line 11).

After choosing a specific number of groups, the *group adjustment* (line 14) tries to reduce load imbalances between the created subsets of cores that might arise from an uneven assignment of workload. The adjustment of group sizes is performed such that subsets with a high computational work get more symbolic cores compared to subsets with a lower computational work. The computational work $T_{seq}(G_l)$ of a subset G_l is defined as

$$T_{seq}(G_l) := \sum_{M_i \in \mathcal{M}_l} T_{comp}(M_i)$$

where $\mathcal{M}_l$ denotes the set of M-tasks assigned to subset G_l. The adjusted number of symbolic cores g_l assigned to subset G_l is computed by

$$g_l := \texttt{round}\left(\frac{T_{seq}(G_l)}{\sum_{j=1}^{g} T_{seq}(G_j)} \cdot p\right), l = 1,\ldots,g.$$

Algorithm 1: Scheduling of the layers of the M-task graph.

```
1  begin
2      foreach (layer W = {M_1, ..., M_k} ) do
3          let C be the set and P = |C| the number of symbolic cores;
4          T_min = ∞;
5          for (g = 1, ..., P}) do
6              partition C into g subsets G = {G_1, ..., G_g} of about equal size;
7              sort {M_1, ..., M_k} such that
8                  T_symb(M_1, ⌊P/g⌋) ≥ ... ≥ T_symb(M_k, ⌊P/g⌋);
9              for (j = 1, ..., k) do
10                 assign M_j to G_l with the smallest accumulated execution time;
11             T_act(g) = max_{1≤j≤g} accumulated execution time of G_j;
12             if (T_act(g) < T_min) then
13                 T_min = T_act(g); g_min = g; G_min = G;
14         group_adjustment(W, g_min, G_min);
```

The rounding is performed such that the total number of symbolic cores, $\sum_{i=1,\ldots,g} g_i$ is equal to the number of physical cores of the target platform.

Figure 6 shows three possible schedules for the extrapolation method. The schedule computed by Alg. 1 depends on the computation and communication performance of the actual target platform. For the platforms used for the benchmark tests in Sect. 4, an execution with $R/2$ subsets of cores (see Fig. 6 (middle)) is selected where R is the number of approximations computed by the extrapolation method.

3.3 Architecture Model

For heterogeneous systems, the specific selection of execution cores used for the M-tasks can have a large influence on the resulting communication and execution time, since different communication costs for internal M-task communication and re-distributions between M-tasks may result. In this article, we focus on multi-core systems as a special form of a heterogeneous platform. We assume cores of the same type but with different interconnections between (i) cores of the same processor, (ii) processors of the same node, and (iii) nodes of a partition of the entire machine.

The architecture can be represented by a tree structure with cores C as leaves,

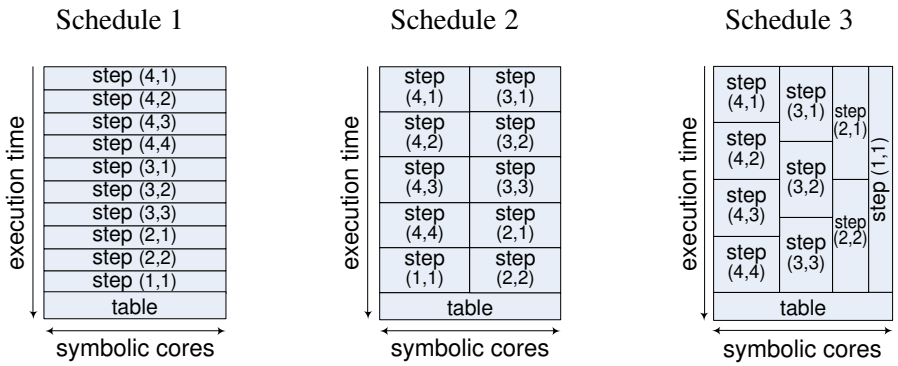

Fig. 6: Three possible schedules for the M-task graph for one time step of the extrapolation method with $R = 4$ approximations from Fig. 5 computed by Alg. 1: (Left) A data parallel execution with $g = 1$ groups. (Middle) A task parallel execution with $g = R/2 = 2$ groups. (Right) A task parallel execution with $g = R = 4$ groups of different size as determined by the subsequent group adjustment step.

processors P as intermediate nodes being a parent for cores, computing nodes N as intermediate nodes combining processors, and partitions or the entire machine A as root node. For a unique identification of the leaf nodes k of the architecture tree, we use the label $l(k) = nid.pid.cid$ consisting of the node id nid, the processor id pid, and the core id cid. Figure 7 shows an illustration. In contrast to the Multi-BSP model [49], the architecture tree is not annotated with performance parameters. These parameters are included in the cost functions for M-tasks, see Sect. 3.1.

3.4 Mapping

The scheduling algorithm partitions a given M-task graph into layers and schedules the M-tasks of one layer onto disjoint groups of symbolic cores. For a given layer W, the scheduling algorithm has computed a group partitioning $G = \{G_1, \ldots, G_g\}$ of symbolic cores where each group may execute several M-tasks one after another.

An architecture-aware mapping of an M-task program to a multi-core system is the assignment of the symbolic cores used by the scheduling algorithm to physical cores of the architecture. Such an assignment has to be determined for each layer of the M-task program. The mapping function F_W for layer W maps each symbolic group G_i to a set of physical cores $F_W(G_i)$ with $|G_i| = |F_W(G_i)|$, $i = 1, \ldots, g$, i.e., each group of symbolic cores is mapped to a physical group of the same size. Moreover, different groups of symbolic cores have to be mapped to disjoint sets of

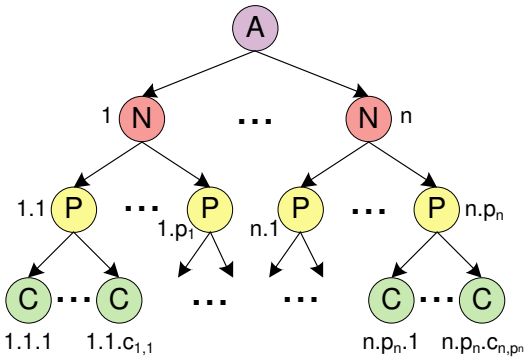

Fig. 7: Tree structure representing a hierarchical multi-core SMP cluster consisting of the entire architecture (A), nodes (N), processors (P) and cores (C). The labels of the nodes are constructed top-down following the path from the root to the respective node.

physical cores, i.e., $F_W(G_i) \cap F_W(G_j) = \emptyset$ for $i, j = 1, \ldots, g$ and $i \neq j$. An illustration of the mapping is shown in Fig. 8.

The definition of the mapping function uses two sequences: a sequence of symbolic cores and a sequence of physical cores. The sequence of symbolic cores

$$sc_{1,1}, \ldots, sc_{1,|G_1|}, sc_{2,1}, \ldots, sc_{g,|G_g|}$$

contains all symbolic cores assigned to the layer in the order of the symbolic groups where $sc_{i,j}$ denotes the j-th symbolic core of group G_i, $i = 1, \ldots, g$, $j = 1, \ldots, |G_i|$. This sequence has been determined by the scheduling algorithm. The P physical cores of the architecture are arranged in a sequence

$$pc_1, pc_2, \ldots, pc_P$$

that contains each physical core exactly once. The order of the physical cores in this sequence is the result of the mapping strategy applied as described below. The mapping function F_W assigns each symbolic core to the physical core with the same position, i.e., the i-th core in the symbolic core sequence is mapped to the i-th physical core in the physical core sequence, $i = 1, \ldots, nc$. For each group G_i, the function F_W is given as:

$$F_W(G_i) = \left\{ pc_j, pc_{j+1}, \ldots, pc_{j+|G_i|-1} \;\middle|\; j = 1 + \sum_{k=1}^{i-1} |G_k| \right\}.$$

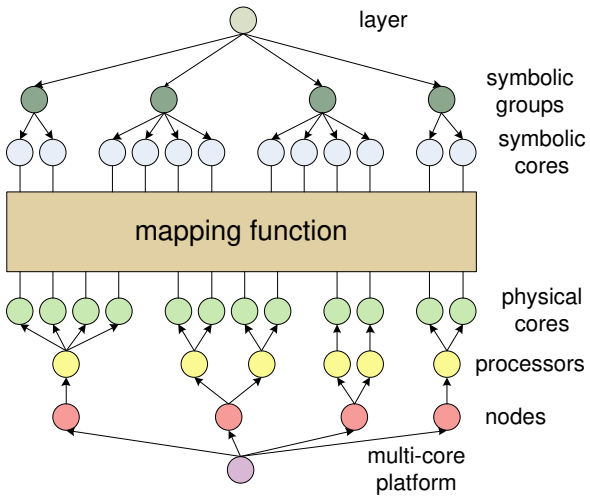

Fig. 8: Illustration of the mapping function from symbolic to physical cores.

We study three different mapping functions.

Consecutive mapping: For the consecutive mapping, the sequence of physical cores is defined such that the cores of the same node are adjacent to each other. Thus, the sequence of physical nodes is defined as

$$1.1.1, \ldots, 1.1.c_{1,1}, 1.2.1, \ldots, 1.p_1.c_{1,p_1}, 2.1.1, \ldots, n.p_n.c_{n,p_n}$$

where n denotes the number of nodes of the platform, p_i is the number of processors of node i,$i = 1, \ldots, n$, and $c_{i,j}$ is the number of cores of processor j of node i, $j = 1, \ldots, p_i$. An example for the consecutive mapping is shown in Fig. 9.

The consecutive mapping tries to minimize the number of physical nodes used for a group of symbolic cores. If a group of symbolic cores is larger than the number of physical cores per node, several consecutive nodes are used. As a result, group internal communication is mainly performed between cores of the same node leading to a reduction of execution time, especially for applications with a high amount of communication within M-tasks. Furthermore, hybrid MPI+OpenMP M-tasks can benefit from the consecutive mapping, since OpenMP threads can be used for the processes mapped to the same physical node.

Scattered mapping: The scattered mapping arranges the physical cores in a sequence such that the corresponding cores of different nodes appear one after another. For a platform with n nodes each consisting of p processors with c cores the sequence of physical nodes is defined as

$$1.1.1, \ldots, n.1.1, 1.1.2, \ldots, n.1.c, 1.2.1, \ldots, n.p.c.$$

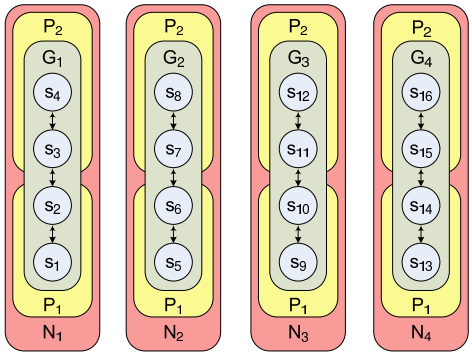

Fig. 9: Illustration of a consecutive mapping of a group partitioning into four symbolic groups $G_1, \ldots, G_4$ each including four symbolic cores on a platform with four nodes, each consisting of two dual-core processors. The edges symbolize communication within M-tasks.

An illustration is shown in Fig. 10.

The scattered mapping tries to assign the symbolic cores of the same group to physical cores located on different nodes. As a result, group-internal communication is mainly performed between cores of different nodes. This might be an advantage if M-tasks with few group-internal communication operations and communication-intensive M-tasks are executed concurrently, since the inter-node communication bandwidth of the platform can then be primarily used by the communication-intensive M-tasks. Additionally, the scattered mapping might restrict the execution of data re-distribution operations between different M-tasks to a single physical node. Especially, the orthogonal communication operations used in the solvers for ODE IVPs considered in Sect. 4.2 can often benefit.

Mixed mapping: Consecutive and scattered mapping strategies can also be mixed. The mixed mapping uses a parameter d denoting the number of consecutive physical cores of a node that are assigned to the same M-task. The sequence of physical nodes starts with the first d cores of the first node, followed by the first d cores of the second node, and so on.

The consecutive and the scattered mapping are special cases of the mixed mapping: the scattered mapping results for $d = 1$, and $d = p_{max} * c_{max}$ leads to a consecutive mapping where p_{max} is the maximum number of processors per node and c_{max} is the maximum number of cores per processor. The parameter d can be used to adapt the mixed mapping to the ratio of intra M-task communication and communication between M-tasks. An illustration of this mapping strategy is given in Fig. 11.

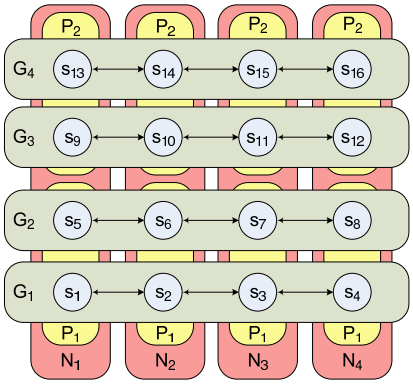

Fig. 10: Illustration of a scattered mapping of symbolic groups $G_1, \ldots, G_4$ with four symbolic cores each on a multi-core platform consisting of four identical nodes each equipped with two dual-core processors.

Figure 12 shows the different mapping strategies applied to the first layer in the schedule of the extrapolation method (see Fig. 6 (middle) for an illustration of the schedule) and for eight nodes of the CHiC cluster (the hardware description is given in Sect. 4.1). The scattered and the mixed ($d = 2$) mapping use the same set of cores to execute the two M-tasks of the layer, however the sequence of physical cores has a different ordering leading to different communication patterns at runtime.

4 Experimental Evaluation

This section describes experimental results obtained by applying the combined scheduling and mapping algorithm to different application programs. The benchmarked program versions are generated by the CM-task compiler [17], which takes the specification of the M-task structure of an application as input, see Fig. 3 for an example. The CM-task compiler carries out the scheduling and mapping, adds the required data re-distribution operations and produces an MPI program that executes the M-tasks on the appropriate subsets of cores.

4.1 Hardware description

For the benchmark tests, three different platforms have been used. The Chemnitz High Performance Linux (**CHiC**) cluster is built up of 530 nodes consisting of two AMD Opteron 2218 dual-core processors with a clock rate of 2.6 GHz. The peak

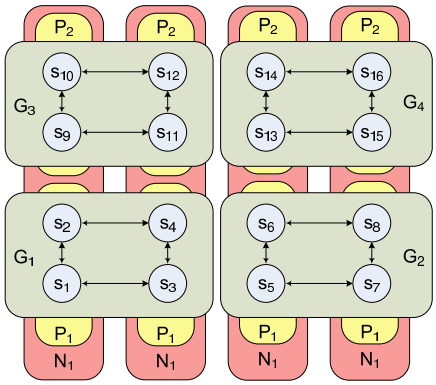

Fig. 11: Illustration of a mixed mapping with $d = 2$ of symbolic groups $G_1, \ldots, G_4$ each consisting of four symbolic cores on a platform with four identical nodes each comprising two dual-core processors.

performance of a single core is 5.2 GFlops/s. The nodes are interconnected by an SDR Infiniband network. For the benchmark tests, the MVAPICH 1.0 MPI library and the Pathscale Compiler 3.1 are used.

The **SGI Altix** system consists of 19 partitions. The benchmarks are executed inside a single partition containing 128 nodes, each equipped with two Intel Itanium2 Montecito dual-core processors. The processors are clocked at 1.6 GHz and achieve a peak performance of 6.4 GFlops/s per core. Each node has two links to the NUMAlink 4 interconnection network with a bidirectional bandwidth of 6.4 GByte/s per link. The MPI library SGI MPT 1.16 and the Intel Compiler 11.0 are used.

The **JuRoPA** cluster consists of 2208 nodes with two Intel Xeon X5570 "'Nehalem"' quad-core processors each. The processors run at 2.93 GHz leading to a peak performance of 11.72 GFlops/s. A QDR Infiniband network connects the nodes. The software configuration includes the ParaStation MPI library v5.0 and the Intel Compiler 11.0.

4.2 Description of the ODE benchmarks

Different numerical solution methods for systems of ODE IVPs are investigated, see Sect. 2.2.3 for the problem definition. These methods include the explicit extrapolation (EPOL), Iterated Runge-Kutta (IRK), and Parallel Adams-Bashforth (PAB) methods as well as the implicit Diagonal-Implicitly Iterated Runge-Kutta (DIIRK) and Parallel Adams-Bashforth-Moulton (PABM) methods. The specifi-

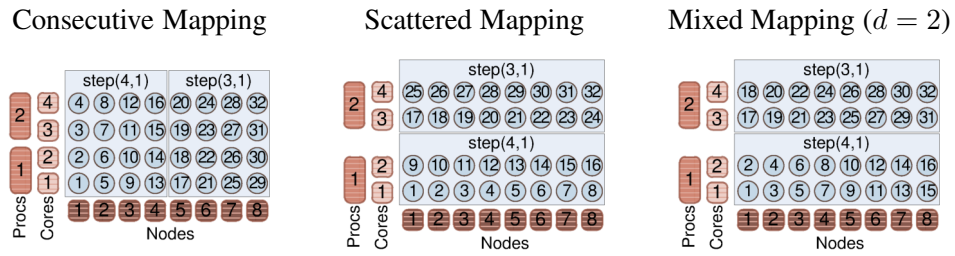

Fig. 12: Illustration of the mapping strategies for the first layer of the schedule for the extrapolation method (from Fig. 6 schedule 2). The numbers in circles indicate the position of the respective cores in the sequence of physical cores.

cation program for the EPOL method is shown in Fig. 3. The IRK, PAB, DIIRK and PABM methods each compute a fixed number K of independent stage vectors within each time step. Coarse grained task parallelism exists between the computations of different stage vectors. Fine grained data parallelism can be used for the individual stage vectors by a parallel computation of their components.

For the benchmark tests, two different program versions are used for each of these ODE solvers. The **data parallel** version represents an implementation that does not exploit any task parallelism, i.e., all M-tasks are computed one after another using all processor cores available. This program version may include many global communication operations.

The **task parallel** program version results from applying the scheduling and mapping algorithm proposed in Sect. 3 to the respective M-task graph. For the EPOL methods, the scheduling algorithm partitions the available cores into $R/2$ equal-sized subsets. Each of the subsets is responsible for the computation of approximation i and $R-i+1, i = 1, \ldots, R/2$, see Fig.6 (middle) for an illustration. Thus, each subset computes the same number of micro steps leading to a good load balance. For the IRK, PAB, DIIRK and PABM methods, the K stage vectors are computed concurrently on K disjoint subsets of cores. The task parallel program version might be more efficient than the data parallel version, since the communication operations within the M-tasks are restricted to subsets of cores leading to smaller communication costs. On the other hand, additional data re-distribution operations are required to exchange intermediate results between the subsets of cores.

The program versions discussed include the following three types of communication operations:

- Global communication operations are executed by all cores available.
- Group-based communication operations are executed by the cores of the subset

Bench-mark	global communication	group-based communication	orthogonal communication
EPOL(dp)	$R(R+1)/2 * T_{ag}$	$-$	$-$
EPOL(tp)	$1 * T_{bc}$	$(R+1) * T_{ag}$	$-$
IRK(dp)	$(K * m + 1) * T_{ag}$	$-$	$-$
IRK(tp)	$1 * T_{ag}$	$m * T_{ag}$	$m * T_{ag}$
DIIRK(dp)	$1 * T_{ag} + K *$ $(n-1) * I * T_{bc}$	$-$	$-$
DIIRK(tp)	$1 * T_{ag}$	$(n-1) * I * T_{bc}$	$m * T_{ag}$
PAB(dp)	$K * T_{ag}$	$-$	$-$
PAB(tp)	$-$	$1 * T_{ag}$	$1 * T_{ag}$
PABM(dp)	$K(1+m) * T_{ag}$	$-$	$-$
PABM(tp)	$-$	$(1+m) * T_{ag}$	$1 * T_{ag}$

Table 1: Types and amount of collective communication operations executed for one time step of the ODE solvers in the data parallel (dp) and in the task parallel (tp) program version. The execution times of a broadcast and a multi-broadcast operation are denoted as T_{bc} and T_{ag}, respectively. The size of the ODE system is denoted as n. The number of fixed point iterations executed are denoted either as m (statically determined) or I (dynamically determined).

assigned to the same M-task, e.g., by the set of cores $\{s_1, s_2, s_3, s_4\}$ in Fig. 9.

- Orthogonal communication operations are used to exchange data between cores with the same position within concurrently executed M-tasks, e.g., by the set of cores $\{s_1, s_5, s_9, s_{13}\}$ in Fig. 9.

Table 1 shows the number of collective communication operations executed within a single time step of the ODE solvers. For the task parallel versions, the communication operations for one of the disjoint groups of cores are listed. The execution time of a broadcast operation (`MPI_Bcast()`) is denoted as T_{bc} and the execution time of a multi-broadcast operation (`MPI_Allgather()`) is denoted as T_{ag}. The size of the ODE system considered is denoted as n, and m defines the number of fixed point iteration steps executed to compute the stage vectors. Both, n and m, are defined statically by the underlying method. The number I of fixed point iterations in the DIIRK method is determined dynamically using a convergence criterion and is typically small, i.e., $1 \leq I \leq 3$ in most cases.

The benchmarks are executed with two types of ODE systems called sparse and dense in the following. The sparse system results from the spatial discretization of the 2D Brusselator equation (BRUSS2D) [21]. The dense system arises from a Galerkin approximation of a Schrödinger-Poisson system (SCHROED) [41]. The time required to evaluate the entire ODE system depends linearly (sparse system)

or quadratically (dense system) on the ODE system size.

4.3 Evaluation of the Scheduling Algorithm

First, we evaluate the scheduling step proposed in Sect. 3.2 by a comparison with the algorithms CPA [39] and CPR [38]. The scheduling algorithms CPA and CPR have been selected because they are often used in comparison benchmarks and form the basis of many other algorithms [4, 16, 52]. Both algorithms consist of an allocation phase and a scheduling phase. The allocation phase assigns every M-task a number of executing cores such that a good trade-off between the reduction of the length of the critical path of the M-task graph and the number of M-tasks that might be executed concurrently is achieved. The scheduling phase determines the execution order and assigns subsets of cores to the M-tasks. CPA decouples the allocation and the scheduling phase from each other whereas CPR repeatedly executes both phases until the currently computed schedule cannot be improved further. In general, the schedules produced by CPA and CPR do not exhibit a layered structure and, thus, these algorithms cannot be combined with the mapping step proposed in Sect. 3.4.

Figure 13 (left) shows the benchmark results obtained for the PABM method with $K = 8$ stage vectors using the consecutive mapping for all program versions. The speedups shown are the quotient of the average execution times of a time step of the sequential and the parallel implementation. The average execution times are determined by executing several hundred time steps. The figure shows that the schedules computed by CPA are not competitive. The reason is the large amount of idle time that results from an "over-allocation" in the allocation phase, i.e., the number of cores assigned to the K independent M-tasks exceeds the number of physically available cores. Thus, the scheduling phase cannot execute all K M-tasks concurrently. CPR computes schedules that are identical with the task parallel version obtained by the scheduling step from Sect. 3.2. Similar results are also obtained for the IRK method with $K = 4$ stage vectors (not shown in a figure).

Figure 13 (right) compares the average execution times of a single time step of the EPOL method using different scheduling decisions for the M-task graph from Fig. 4. For this method, CPA computes a mixed parallel schedule that leads to low execution times. CPR tries to reduce the length of the critical path in the M-task graph by assigning a large number of cores to the M-tasks in the longest linear chain leading to an almost data parallel execution of these M-tasks. This decision leads to high communication costs within these M-tasks. Due to the additional data re-distribution operations, a higher overall execution time than a pure data parallel execution results. These results show that the structure of the M-task graph greatly influences the performance of the schedules produced by CPA and CPR. In contrast

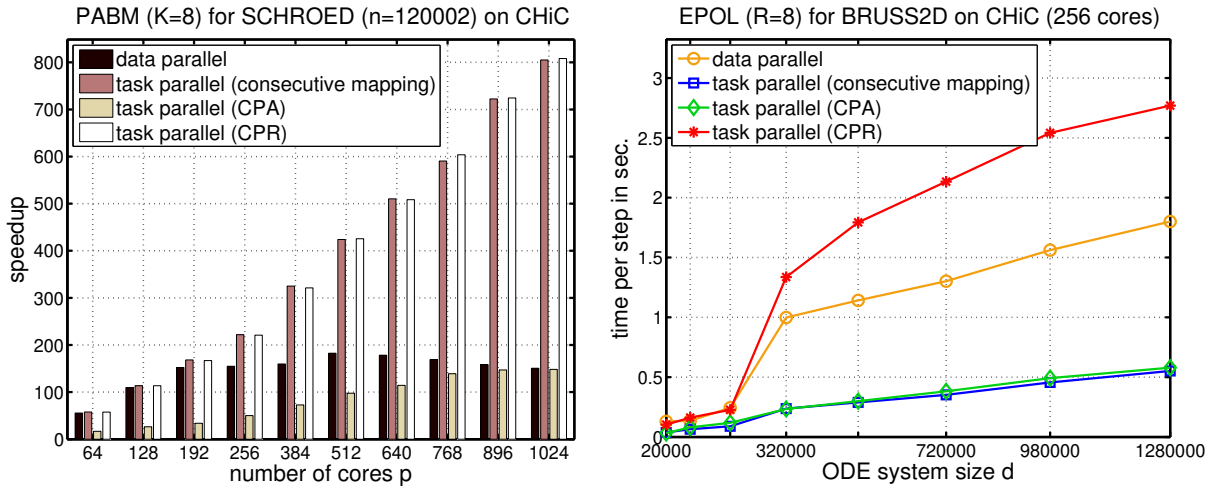

Fig. 13: Comparison of the performance of different scheduling decisions for the PABM method with $K = 8$ stage vectors (left) and the EPOL method with $R = 8$ approximations (right) on the CHiC cluster.

to the scheduling algorithm from Sect. 3.2, none of these two algorithms computes good schedules for all benchmarks considered.

4.4 Evaluation of the Mapping Strategies for Collective Communication Operations

In the following, we investigate the impact of different mapping strategies on the performance of a multi-broadcast operation (`MPI_Allgather()`). This collective communication operation has been selected because it is required by all ODE benchmarks and influences the communication times in these benchmarks considerably. Figure 14 (left) shows the execution time of a global multi-broadcast operation on 256 cores of the CHiC cluster. The results show that a consecutive mapping clearly leads to the lowest execution times. This behavior can be attributed to ring-based algorithm used by the MPI library for large messages leading to communication between processes with neighboring ranks. Using a consecutive mapping, this communication occurs primarily within the nodes of the cluster.

The performance of the group-based and orthogonal communication operations is assessed with the Multi-Allgather benchmark of the Intel MPI benchmark suite [22]. This benchmark creates a fixed number of equal-sized subsets of cores and concurrently executes multi-broadcast operations within each group. Figure 14 (right) shows the results for this benchmark on the CHiC cluster using four groups

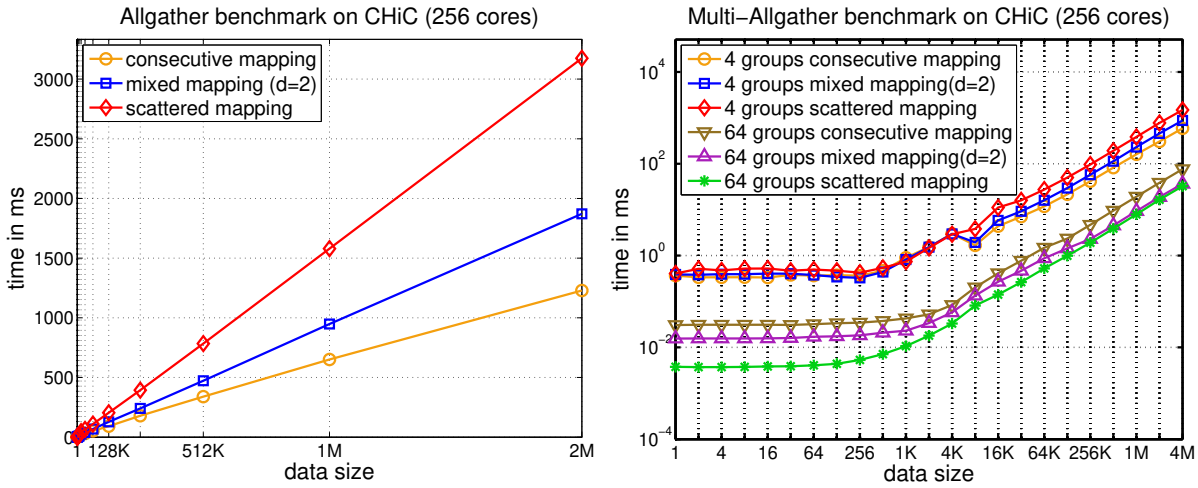

Fig. 14: (Left) Measured execution times for an `MPI_Allgather()` operation on 256 cores of the CHiC cluster. (Right) Execution time for the Multi-Allgather benchmark with 64 groups each including four cores and four groups each including 64 cores and different processor placements. The data size refers to the amount of data provided by each participating core.

and 64 groups, respectively. For an ODE solver with $K = 4$ stage vectors, the case with four groups corresponds to the group-based communication and the 64 groups case corresponds to the orthogonal communication. The results show that group-based communication benefits from a consecutive mapping whereas the scattered mapping leads to the lowest runtimes for the orthogonal communication.

4.5 Evaluation of the Mapping Strategies for ODE Solvers

In the following, we focus on the mapping step and compare different mapping decisions for the ODE benchmarks. Since the data parallel program version only uses global collective communication operations, the consecutive mapping achieves the highest performance for all benchmarks considered. So, in the following, we concentrate on the impact of the mapping strategies on the performance of the task parallel program version.

Figure 15 shows the execution times of a single time step of the IRK method with $K = 4$ stage vectors applied to the Brusselator system on the JuRoPA and CHiC clusters in the top row. The CHiC cluster contains four cores per node and, thus, a consecutive, a scattered, and a mixed ($d = 2$) mapping are considered. On the JuRoPA cluster, additionally a mixed ($d = 4$) mapping is used, since this

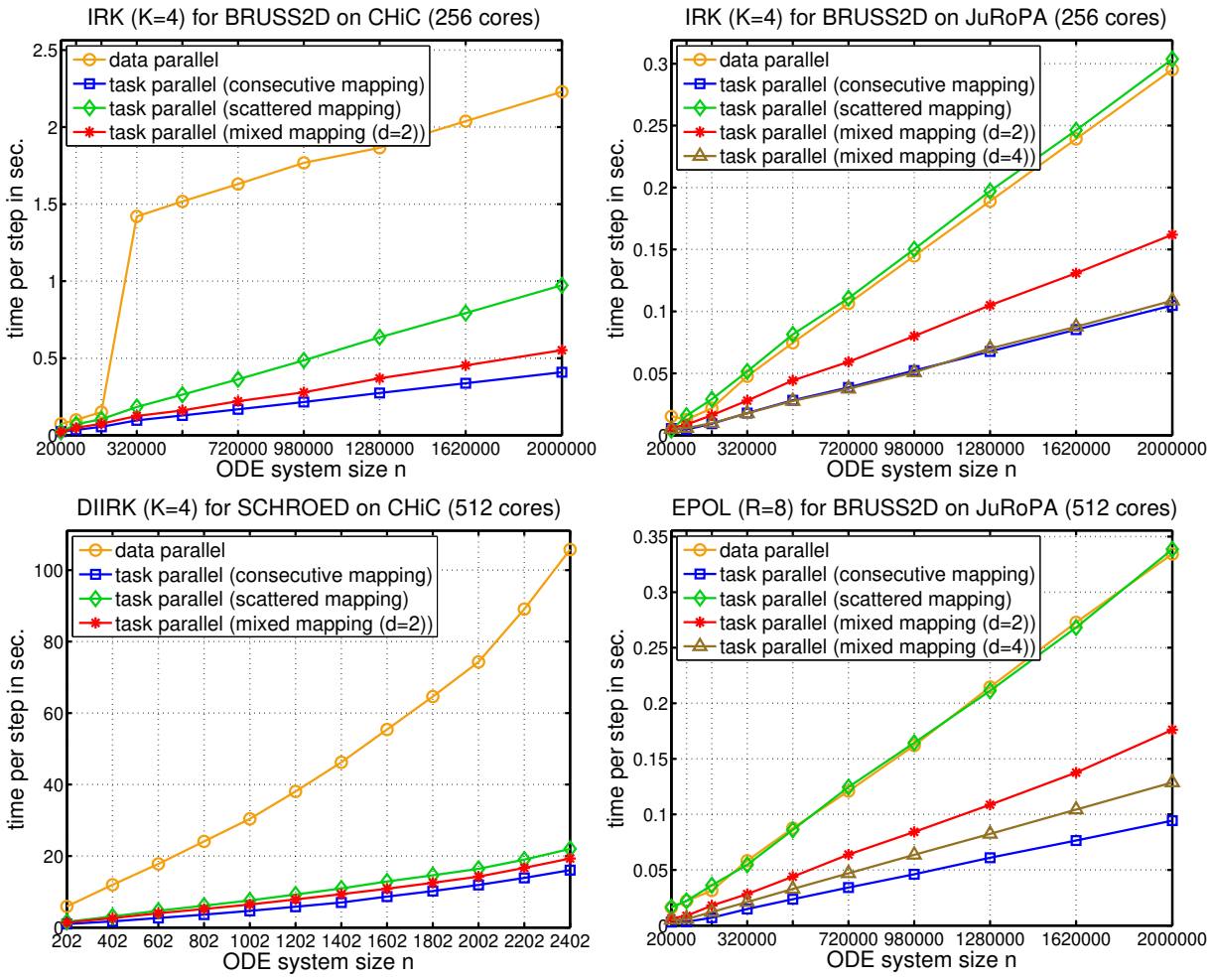

Fig. 15: Measured execution times for a single time step of the IRK (top row), DIIRK (bottom left), and EPOL (bottom right) methods on the CHiC cluster (left column) and the JuRoPA cluster (right column).

platform contains eight cores per node. On both platforms the lowest execution times are achieved by mapping as many symbolic cores of a group as possible onto the same cluster node. A scattered mapping is clearly outperformed by the other mappings. This behavior mainly results from the global communication operations in the IRK method.

Compared to the IRK method, the DIIRK method includes much more communication within the individual M-tasks, which can be restricted to subsets of cores by a task parallel execution scheme. Therefore, the task parallel version achieves much lower execution times compared to pure data parallelism as it is shown in Fig. 15 (bottom left) for 512 cores of the CHiC cluster. Also for the DIIRK method, the lowest execution times are achieved by a consecutive mapping due to the global communication operations.

The consecutive mapping is also beneficial for the EPOL method as it is shown for 512 cores of the JuRoPA cluster and $R = 8$ approximations in Fig. 15 (bottom right). The mixed mapping ($d = 4$) leads to a substantially higher execution time than the consecutive mapping. This can be attributed to the absence of orthogonal communication operations in the EPOL method.

Figure 16, top row, shows the measured execution times for a single time step of the PAB method. The task parallel version of this benchmark includes an equal number of group-based and orthogonal communication operations. Therefore, a mixed mapping strategy with $d = 2$ (CHiC cluster) or with $d = 4$ (JuRoPA cluster) leads to the lowest execution times.

The PABM method has more computation and communication within the M-tasks than the PAB method. Therefore, a placement of the processes executing the same M-task on the same cluster node is desirable. The speedups obtained for the dense system on the CHiC cluster shown in Fig. 16 (bottom left) confirm this observation. For a high number of processor cores, the consecutive mapping of the task parallel execution scheme is clearly superior to the other program versions. The scalability of the data parallel version is limited to 512 processor cores because of the high amount of global collective communication. The runtimes of the sparse system on the JuRoPA cluster that are presented in Fig. 16 (bottom right) show a similar behavior, i.e., the consecutive mapping leads to the lowest runtimes, and data parallelism is outperformed by all task parallel versions.

Combining these observations, it can be seen that the consecutive mapping should be selected for ODE solvers with either dominating group-based communication (EPOL, DIIRK, and PABM methods) or global communication patterns (IRK and DIIRK methods). For these methods, the consecutive mapping leads to communication primarily within cluster nodes. On the other hand, for algorithms with an equal amount of group-based and orthogonal communication (PAB method) the mixed mapping leads to the best results.

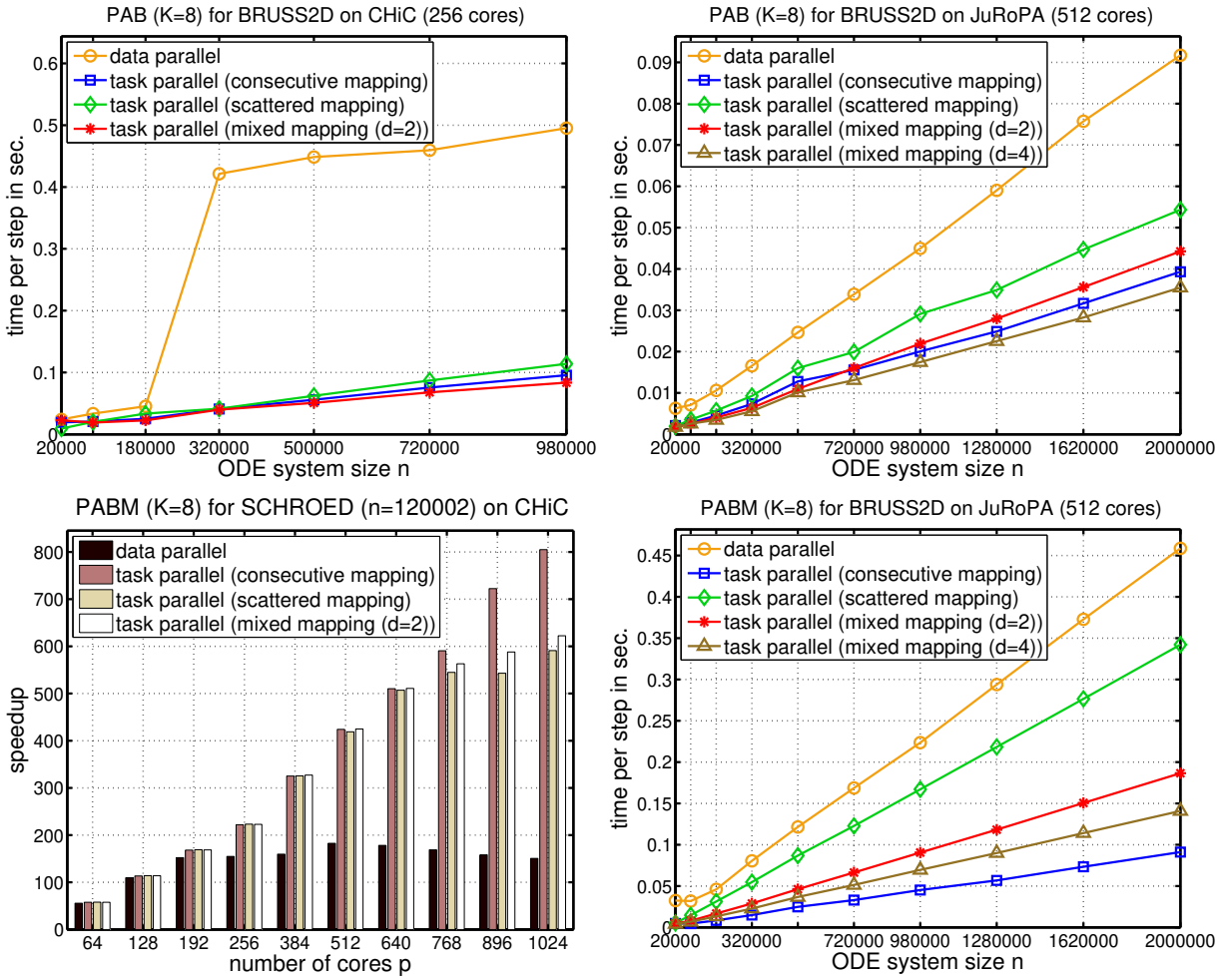

Fig. 16: Benchmark results for the PAB (top) and PABM (bottom) methods with $K = 8$ stage vectors on the CHiC cluster (left) and the JuRoPA cluster (right).

4.6 Evaluation for the NAS benchmarks

Another class of applications that can benefit from the M-task programming model are solvers for flow equations operating on a set of meshes (also called zones). Within each time step, the computation of the solution is performed independently for each zone. At the end of a time step, a border exchange between overlapping zones is required. The NAS parallel benchmark multi-zone version (NPB-MZ) provides solvers for discretized versions of the unsteady, compressible Navier-Stokes equations that operate on multiple zones [50]. The reference implementation uses shared memory OpenMP programming to exploit the fine grained parallelism within the zones and message passing with MPI to implement the coarse grained parallelism between different zones. Thus, this implementation restricts the computation of each zone to a shared memory environment, e.g., a node of a multi-core cluster. For the purpose of this article, we consider modified versions of the SP-MZ (Scalar Pentagonal Multi-Zone) and BT-MZ (Block Tridiagonal Multi-Zone) benchmarks, which use different solvers (SP or BT) to compute discrete solutions in three spatial dimensions. Both versions use MPI for both levels of parallelism and, thus, do not restrict the scheduling and mapping decisions. Each zone is represented by an M-task, leading to z independent M-tasks for z zones. Point-to-point communication is used for both, communication within M-tasks and the border exchanges between M-tasks.

Figure 17 shows the performance results for the SP-MZ benchmark for the CHiC cluster (top left) and for the SGI Altix (top right). The figure compares different scheduling decisions for a fixed number of cores, i.e., different selections for the number of subsets of symbolic cores created (see line 5 of Alg. 1). Benchmark classes C and D with 256 and 1024 equal sized zones are considered. The figure shows that an exploitation of the maximum degree of the available task parallelism, i.e., building 1024 groups for class D and 256 groups for class C, does not lead to the highest performance. On the CHiC cluster, the best execution scheme results by using 64 parallel groups, assigning 16 neighboring zones to each group and using a scattered mapping. For the SGI Altix, the program version with 128 parallel groups leads to the best performance. Again, a scattered mapping strategy outperforms the other program versions. The program versions with a low number of groups are not competitive because each M-task is executed by a large number of processor cores, leading to a high communication and synchronization overhead within the groups.

The zones of the BT-MZ benchmark incorporate different amounts of computation and, thus, the assignment of M-tasks to subsets of cores and load balancing between different subsets of cores becomes an important issue. The performance results achieved for a varying number of parallel groups are shown in Fig. 17 for

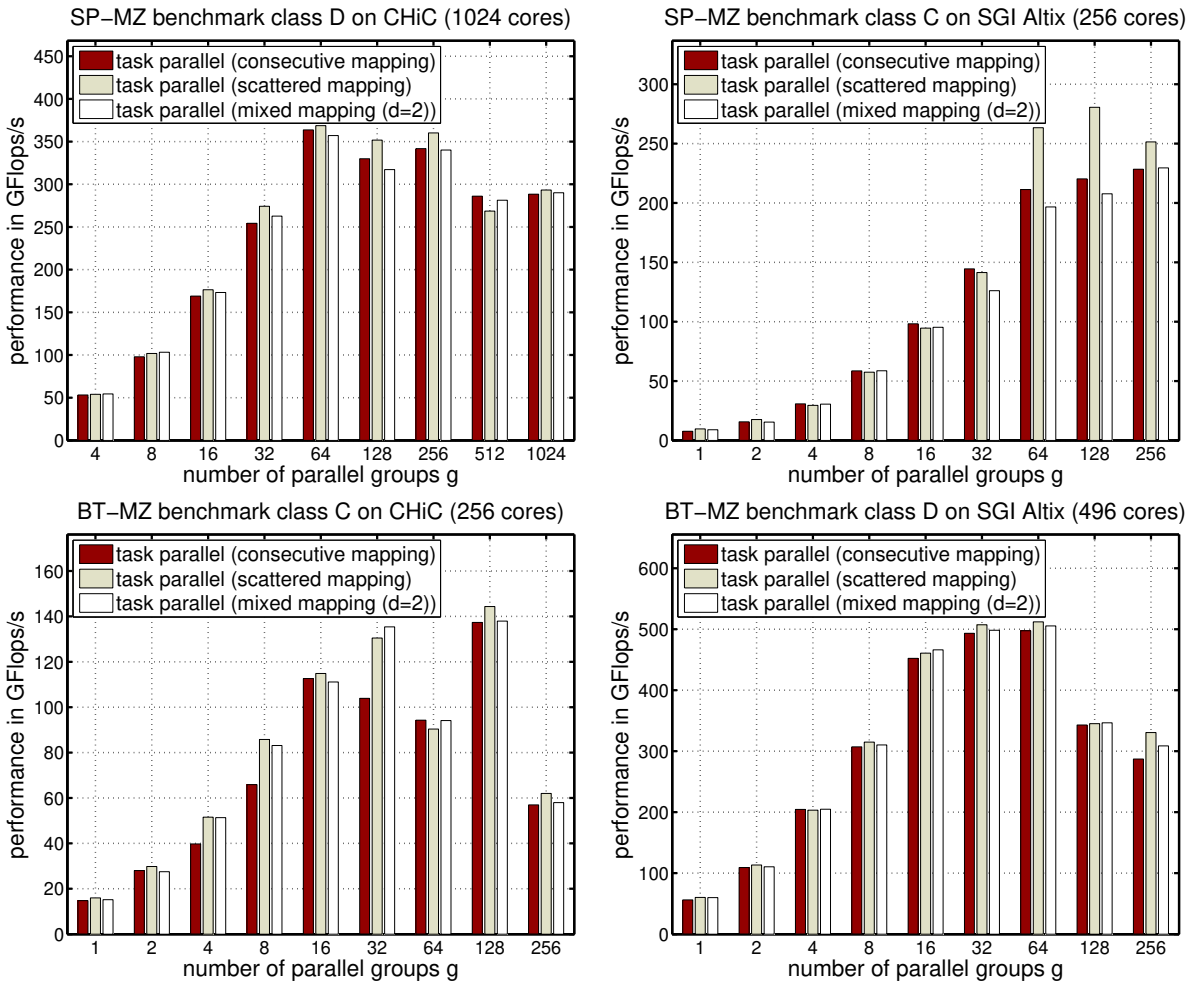

Fig. 17: Performance of the NAS benchmarks SP-MZ (top) and BT-MZ (bottom) executed on the CHiC cluster (left) and the SGI Altix (right) using different numbers of disjoint subsets of cores.

class C with 256 zones on the CHiC cluster (bottom left) and for class D with 1024 zones on the SGI Altix (bottom right). On the CHiC cluster, the highest performance is obtained by the execution schemes with 32 and 128 subsets of cores. For the SGI Altix, the creation of 32 and 64 subsets of cores leads to the best results. On both platforms, the scattered mapping outperforms the other mapping strategies. The performance obtained for the execution schemes with a large number of subsets of cores is impacted by load imbalances. These load imbalances exist because the M-tasks have different execution times and only one or a few M-tasks are assigned to a specific subset of cores.

4.7 MPI Tasks vs. OpenMP Threads

An adaption to the hardware characteristics of clusters of SMPs can be achieved by combining message passing programming using MPI and thread programming using OpenMP into hybrid programming models. In this section, we examine the performance of hybrid realizations of M-task programs. The upper level parallelism between M-tasks is realized by MPI communication and for the lower level parallelism within M-tasks hybrid MPI+OpenMP implementations are used. Multiple symbolic cores of the same M-task have to be mapped on the same cluster node to make use of OpenMP parallelism. Therefore, a suitable mapping strategy is required. In the following, we focus on a consecutive mapping for both, pure MPI and hybrid implementations.

Figure 18 shows a comparison of the speedups achieved for the IRK method with $K = 4$ stages on the CHiC cluster using four OpenMP threads per cluster node. The hybrid execution scheme for the data parallel version leads to considerable higher speedups compared to a pure MPI realization. The main reason for this improvement is the reduction of the number of MPI processes participating in global communication operations.

Figure 18 (right) shows the execution times of one time step of the DIIRK method with four stages on the CHiC cluster. The hybrid execution leads to a slowdown for the data parallel version caused by program parts that require a frequent synchronization. For the task parallel version, the hybrid execution scheme clearly outperforms its pure MPI counterpart.

Since the SGI Altix has a distributed shared memory architecture that allows the use of OpenMP threads across different nodes, many different combinations of MPI processes and OpenMP threads are possible. Figure 19 shows a comparison of the execution times of the PABM method with $K = 8$ stages on 256 cores of the SGI Altix. At least eight MPI processes are required for the task parallel version, one for each stage. Using 256 OpenMP threads leads to the best results for the data parallel version. For the task parallel version, the lowest execution times are

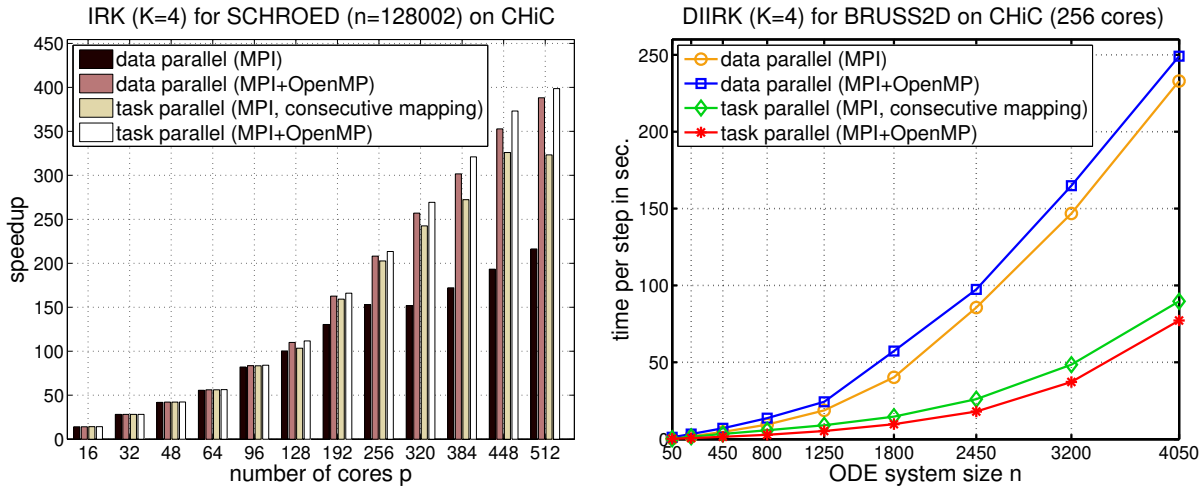

Fig. 18: Comparison of the performance of a pure MPI implementation with a hybrid MPI+OpenMP realization of the IRK method (left) and the DIIRK method (right) both with $K = 4$ stages on the CHiC cluster.

achieved by using 64 MPI processes and 4 OpenMP threads, i.e., running one MPI process on each of the nodes.

5 Related Work

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, M-tasks). S-tasks are often used for program development in shared address spaces, including single multi-core processors. Examples for such approaches are the task concepts in OpenMP 3.0 [33], Cilk [20] and SMPSs [37], FG [12] for out of core algorithms, or the TPL library for .NET [26].

For mixing task and data parallelism as it is supported by the M-task programming model, several approaches have been proposed, including language extensions (such as Fortran M, Opus [10], Braid [53], Fx [48], HPF 2.0, Orca [6]) as well as skeleton-based (such as P3L [35], LLC [15], Assist [51], Lithium [2], DIP [13], SBASCO [14]), library-based, and coordination-based approaches. Most of these approaches are suitable to define a parallel M-task structure and to extract the available degree of task parallelism in form of a task graph, which is the input required for the scheduling and mapping algorithm described in this article.

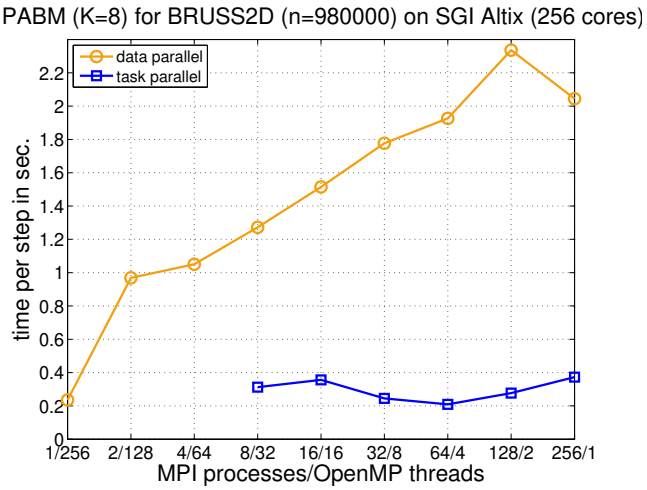

Fig. 19: Parallel runtimes for different combinations of MPI processes and OpenMP threads of PABM with $K = 8$ stages on the SGI Altix.

The scheduling of M-tasks on homogeneous target platforms has been investigated by many research groups. A popular approach is to separate the allocation that fixes the number of executing cores for the M-tasks from the scheduling that computes the execution order and assigns subsets of cores to the M-tasks. Examples for such algorithms are TSAS [40], CPA [39], CPR [38], MCPA [4], and Loc-MPS [52]. Algorithms following this approach are also available with an approximation ratio of 4.73 [23] and, for SP-graphs and task graphs with bounded width, of 2.62 [27]. An alternative approach is to partition the task graph into layers of independent tasks and schedule the resulting layers one after another. In this article, the layers are constructed by a greedy algorithm, but it is also possible to base the layering on the length of the critical path [47]. For the scheduling of a single layer, any scheduling algorithm for independent M-tasks can be used. Examples for such algorithms are the 1.5-approximation algorithm [31], or the algorithms for convex and concave speedup functions [7]. None of the algorithms mentioned above takes the mapping of processes to cores of a multi-core cluster into account. The benchmark results presented in this article show that for multi-core clusters also the mapping of processes to cores may have a large influence on the performance obtained. Therefore, the combination of scheduling and mapping presented is a step forward towards a better exploitation of the performance of such platforms.

Scheduling algorithms for M-tasks on heterogeneous platforms mainly target large cluster-of-clusters systems, see e.g. [8, 19, 32]. These approaches restrict

the execution of an M-task to a single homogeneous sub-cluster. The multi-core clusters used in this article can be regarded as special cluster-of-clusters where each node forms a homogeneous subcluster. But the benchmark results from Sect. 4 show that multiple cluster nodes have to be used to execute the M-tasks in order to obtain a high performance. Therefore, these heterogeneous scheduling algorithms do not seem to be suitable for multi-core clusters. In contrast, our approach can determine a suitable task layout for these clusters. A scheduling algorithm for M-tasks on a heterogeneous cluster with a homogeneous interconnection has been presented in [5].

Mapping techniques for parallel applications try to increase their performance by placing processes with high communication requirements on physical computing units that are connected by a high-speed interconnect. Both, the communication requirements of the considered application and the communication performance of the target platform, can be represented by undirected, weighted graphs. An optimized process placement can be computed by mapping the application graph onto the platform graph and taking into account the assigned weight values. A special graph library is used for solving the mapping problem for multi-core target platforms in [30]. In [45] a recursive doubling heuristic has been used for this purpose. MPIPP [11] is a tool set consisting of components that can obtain the communication profile of an MPI application, determine the network topology of SMP clusters and compute optimized process placements based on a heuristic mapping algorithm. The mapping of task parallel applications on large platforms with different network topologies is examined in [1]. First, graph partitioning is used to assign heavily communicating tasks to the same physical processing unit. In the second step, the computed graph partitions are mapped to the target platform by a heuristic that tries to reduce the number of network hops between communicating tasks. A random search technique is used in [34] to map the processes of data parallel applications on target platforms with switch-based networks. In contrast to our work, the mapping approaches mentioned above do not explicitly support mixed task and data parallel applications and dependencies between processes.

The mapping of a set of tasks each consisting of a fixed number of threads on multi-core platforms has been studied in [29]. The algorithm takes the communication requirements between the tasks into account and ensures that threads belonging to the same task are allocated to the same cluster node. Dependencies between tasks and the determination of an appropriate number of execution cores for the tasks are not considered as it is done by our M-task approach.

6 Conclusions

In this article, we have discussed the suitability of the programming approach with parallel tasks for hierarchical multi-core cluster systems. In particular, we have presented a combined scheduling and mapping approach for programs based on parallel tasks. The scheduling algorithm creates layers of independent parallel tasks, partitions the symbolic cores representing the target architecture into subsets, assigns the parallel tasks of a layer to these subsets, and adjusts the subset sizes according to the computational work. Several strategies are proposed for the mapping step that assigns each symbolic core to a different physical core. These strategies include a consecutive mapping of processes of the same parallel tasks to the same cluster node to increase the performance of task-internal communication and a scattered mapping that assigns processes of different parallel tasks to the same cluster node to improve the communication performance for data exchanges between parallel tasks.

Benchmark tests with several large applications from scientific computing show that the approach based on parallel tasks is a suitable programming model for multi-core clusters, but significant differences in the performance of different mappings can occur. The best mapping depends on both, the communication requirements of the applications and the communication performance of the target platform. For solvers for systems of ordinary differential equations, a consecutive placement of the processes of the same parallel task onto the same cluster node leads to the best results in most cases. The multi-zone benchmarks from the NAS parallel benchmark suite require a careful selection of an appropriate number of subsets of cores as well as a suitable mapping strategy. In the experimental evaluation, the best performance has been obtained by selecting a medium number of subsets of cores and by using a scattered placement of the symbolic cores.

References

[1] T. Agarwal, A. Sharma, and L. V. Kalé. Topology-aware Task Mapping for Reducing Communication Contention on Large Parallel Machines. In *Proc. of the 20th Int. Parallel & Distributed Processing Symp. (IPDPS'06)*, Washington, DC, USA, 2006. IEEE Computer Society.

[2] M. Aldinucci, M. Danelutto, and P. Teti. An Advanced Environment Supporting Structured Parallel Programming in Java. *Future Gener. Comp. Sy.*, 19(5):611–626, 2003.

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

[4] S. Bansal, P. Kumar, and K. Singh. An Improved Two-step Algorithm for Task and Data Parallel Scheduling in Distributed Memory Machines. *Parallel Comput.*, 32(10):759–774, 2006.

[5] J. Barbosa, C. Morais, R. Nobrega, and A.P. Monteiro. Static Scheduling of Dependent Parallel Tasks on Heterogeneous Clusters. In *Proc. of the Int. Conf. on Cluster Computing (CLUSTER'05)*, Washington, DC, USA, 2005. IEEE Computer Society.

[6] S. Ben Hassen, H.E. Bal, and C.J.H. Jacobs. A Task- and Data-parallel Programming Language Based on Shared Objects. *ACM T. Progr. Lang. Sys.*, 20(6):1131–1170, 1998.

[7] J. Blazewicz, M. Machowiak, J. Weglarz, M.Y. Kovalyov, and D. Trystram. Scheduling Malleable Tasks on Parallel Processors to Minimize the Makespan. *Ann. Oper. Res.*, 129(1-4):65–80, 2004.

[8] V. Boudet, F. Desprez, and F. Suter. One-Step Algorithm for Mixed Data and Task Parallel Scheduling without Data Replication. In *Proc. of the 17th Int. Parallel & Distributed Processing Symp. (IPDPS'03)*, Washington, DC, USA, 2003. IEEE Computer Society.

[9] K. Burrage. *Parallel and Sequential Methods for Ordinary Differential Equations*. Clarendon Press, New York, NY, USA, 1995.

[10] B.M. Chapman, M. Haines, P. Mehrota, H.P. Zima, and J. Van Rosendale. Opus: A Coordination Language for Multidisciplinary Applications. *Sci. Program.*, 6(4):345–362, 1997.

[11] H. Chen, W. Chen, J. Huang, B. Robert, and H. Kuhn. MPIPP: An Automatic Profile-guided Parallel Process Placement Toolset for SMP Clusters and Multiclusters. In *Proc. of the 20th Int. Conf. on Supercomputing (ICS'06)*, pages 353–360, New York, NY, USA, 2006. ACM.

[12] E.R. Davidson and T.H. Cormen. Building on a Framework: Using FG for More Flexibility and Improved Performance in Parallel Programs. In *Proc. of the 19th Int. Parallel & Distributed Processing Symp. (IPDPS'05)*, pages 54–63, Washington, DC, USA, 2005. IEEE Computer Society.

[13] M. Díaz, B. Rubio, E. Soler, and J.M. Troya. Domain Interaction Patterns to Coordinate HPF Tasks. *Parallel Comput.*, 29(7):925–951, 2003.

[14] 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, Washington, DC, USA, 2004. IEEE Computer Society.

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

[16] J. Dümmler, R. Kunis, and G. Rünger. A Comparison of Scheduling Algorithms for Multiprocessortasks with Precedence Constraints. In *Proc. of the High Performance Computing & Simulation Conf. (HPCS'07)*, pages 663–669, Prague, Czech Republic, 2007. ECMS.

[17] J. Dümmler, T. Rauber, and G. Rünger. Communicating Multiprocessor-Tasks. In *Proc. of the 20th Int. Workshop on Languages and Compilers for Parallel Computing (LCPC 2007)*, volume 5234 of *LNCS*, pages 292–307, Berlin, Heidelberg, 2007. Springer-Verlag.

[18] J. Dümmler, T. Rauber, and G. Rünger. Scalable Computing with Parallel Tasks. In *Proc. of the IEEE Workshop on Many-Task Computing on Grids and Supercomputers (MTAGS'09)*, New York, NY, USA, 2009. ACM.

[19] P.-F. Dutot, T. N'Takpe, F. Suter, and H. Casanova. Scheduling Parallel Task Graphs on (Almost) Homogeneous Multicluster Platforms. *IEEE T. Parall. Distr.*, 20(7):940–952, 2009.

[20] M. Frigo, C.E. Leiserson, and K.H. Randall. The Implementation of the Cilk-5 Multithreaded Language. *SIGPLAN Notices*, 33:212–223, May 1998.

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

[22] Intel MPI Benchmark. `http://software.intel.com/en-us/articles/intel-mpi-benchmarks/`.

[23] K. Jansen and H. Zhang. An Approximation Algorithm for Scheduling Malleable Tasks under General Precedence Constraints. *ACM T. Algorithms*, 2(3):416–434, 2006.

[24] C.W. Kessler and W. Löwe. A Framework for Performance-Aware Composition of Explicitly Parallel Components. In *Proc. of the Int. Conf. ParCo 2007*,

volume 15 of *Advances in Parallel Computing*, pages 227–234, Amsterdam, The Netherlands, 2007. IOS Press.

[25] M. Kühnemann, T. Rauber, and G. Rünger. Performance Modelling for Task-Parallel Programs. In M. Gerndt, V. Getov, A. Hoisie, A. Malony, and B. Miller, editors, *Performance Analysis and Grid Computing*, pages 77–91. Kluwer Academic Publishers, Norwell, MA, USA, 2004.

[26] D. Leijen, W. Schulte, and S. Burckhardt. The Design of a Task Parallel Library. In *Proc. of the 24th Conf. on Object Oriented Programming Systems Languages and Applications (OOPSLA'09)*, pages 227–242, New York, NY, USA, 2009. ACM.

[27] R. Lepere, D. Trystram, and G.J. Woeginger. Approximation Algorithms for Scheduling Malleable Tasks Under Precedence Constraints. *Int. J. Found. Comput. S.*, 13(4):613–627, 2002.

[28] J.Y.-T. Leung, editor. *Handbook of Scheduling: Algorithms, Models, and Performance Analysis*. CRC Press, Inc., Boca Raton, FL, USA, 2004.

[29] Y. Liu, X. Zhang, H. Li, and D. Qian. Allocating Tasks in Multi-core Processor based Parallel Systems. In *Proc. of the Network and Parallel Computing Workshops (NPC'07)*, pages 748–753, Washington, DC, USA, 2007. IEEE Computer Society.

[30] G. Mercier and J. Clet-Ortega. Towards an Efficient Process Placement Policy for MPI Applications in Multicore Environments. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 5759 of *LNCS*, pages 104–115, Berlin, Heidelberg, 2009. Springer-Verlag.

[31] G. Mounie, C. Rapine, and D. Trystram. A $\frac{3}{2}$-Approximation Algorithm for Scheduling Independent Monotonic Malleable Tasks. *SIAM J. Comput.*, 37(2):401–412, 2007.

[32] T. N'takpé, F. Suter, and H. Casanova. A Comparison of Scheduling Approaches for Mixed-Parallel Applications on Heterogeneous Platforms. In *Proc. of the 6th Int. Symp. on Parallel and Distributed Computing*, Washington, DC, USA, 2007. IEEE Computer Society.

[33] *OpenMP Application Program Interface, Version 3.0*. www.openmp.org, May 2008.

[34] J. M. Orduna, F. Silla, and J. Duato. On the Development of a Communication-aware Task Mapping Technique. *J. Syst. Architect.*, 50(4):207–220, 2004.

[35] S. Pelagatti. Task and Data Parallelism in P3L. In F.A. Rabhi and S. Gorlatch, editors, *Patterns and Skeletons for Parallel and Distributed Computing*, pages 155–186. Springer-Verlag, London, UK, 2003.

[36] S. Pelagatti and D.B. Skillicorn. Coordinating Programs in the Network of Tasks Model. *J. Syst. Integr.*, 10(2):107–126, 2001.

[37] J.M. Perez, R.M. Badia, and J. Labarta. A Dependency-Aware Task-Based Programming Environment for Multi-Core Architectures. In *Proc. of the Int. Conf. on Cluster Computing (CLUSTER'08)*, pages 142–151, Washington, DC, USA, 2008. IEEE Computer Society.

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

[39] A. Radulescu and A.J.C. van Gemund. A Low-Cost Approach towards Mixed Task and Data Parallel Scheduling. In *Proc. of the Int. Conf. on Parallel Processing (ICPP'01)*, pages 69–76, Washington, DC, USA, 2001. IEEE Computer Society.

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

[41] T. Rauber and G. Rünger. Parallel Solution of a Schrödinger-Poisson System. In *Proc. of the Int. Conf. on High-Performance Computing and Networking (HPCN'95)*, volume 919 of *LNCS*, pages 697–702, London, UK, 1995. Springer-Verlag.

[42] T. Rauber and G. Rünger. Compiler Support for Task Scheduling in Hierarchical Execution Models. *J. Syst. Architect.*, 45(6-7):483–503, 1998.

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

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

[45] E.R. Rodrigues, F.L. Madruga, P.O.A. Navaux, and J. Panetta. Multi-core Aware Process Mapping and its Impact on Communication Overhead of Parallel Applications. In *Proc. of the 14th Symp. on Computers and Communications (ISCC'09)*, pages 811–817, Washington, DC, USA, 2009. IEEE Computer Society.

[46] S. K. Sahni. Algorithms for Scheduling Independent Tasks. *J. ACM*, 23(1):116–127, 1976.

[47] R. Sakellariou and H. Zhao. A Hybrid Heuristic for DAG Scheduling on Heterogeneous Systems. In *Proc. of the 13th Heterogeneous Computing Workshop (HCW'04)*, pages 111–123, Washington, DC, USA, 2004. IEEE Computer Society.

[48] J. Subhlok and B. Yang. A New Model for Integrated Nested Task and Data Parallel Programming. *SIGPLAN Not.*, 32:1–12, June 1997.

[49] L.G. Valiant. A Bridging Model for Multi-core Computing. In *Proc. of the 16th European Symp. on Algorithms (ESA'08)*, pages 13–28, Berlin, Heidelberg, 2008. Springer-Verlag.

[50] 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.

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

[52] N. Vydyanathan, S. Krishnamoorthy, G.M. Sabin, U.V. Catalyurek, T. Kurc, P. Sadayappan, and J.H. Saltz. An Integrated Approach to Locality-Conscious Processor Allocation and Scheduling of Mixed-Parallel Applications. *IEEE T. Parall. Distr.*, 20(8):1158–1172, 2009.

[53] E.A. West and A.S. Grimshaw. Braid: Integrating task and data parallelism. In *Proc. of the 5th Symp. on the Frontiers of Massively Parallel Computation (FRONTIERS'95)*, pages 211–219, Washington, DC, USA, 1995. IEEE Computer Society.