Scalable Computing with Parallel Tasks

Jörg Dümmler
Chemnitz University of Technology
Dept. of Computer Science
Straße der Nationen 62
09111 Chemnitz, Germany
djo@cs.tu-chemnitz.de

Thomas Rauber
Bayreuth University
Angewandte Informatik II
Universitätsstr. 30
95447 Bayreuth, Germany
rauber@uni-bayreuth.de

Gudula Rünger
Chemnitz University of Technology
Dept. of Computer Science
Straße der Nationen 62
09111 Chemnitz, Germany
ruenger@cs.tu-chemnitz.de

ABSTRACT

Recent and future parallel clusters and supercomputers use 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 programs and by structuring the application in a way which is sensitive to the heterogeneous interconnect.

In this article, we present an approach to structure the computations of an application as parallel tasks which can interact with other parallel tasks in communication phases. In particular, we consider how these parallel tasks can be mapped onto the computing resources provided by parallel clusters or supercomputers. We show that the scalability can be significantly improved by a suitable task-based organization and a corresponding structuring of the communication within tasks as well as between tasks. We evaluate the impact of different mappings of tasks to cores for different application programs on a variety of parallel machines.

Categories and Subject Descriptors

D1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel Programming*; D4.8 [**Performance**]: Measurements; D4.1 [**Operating Systems**]: Process Management—*Scheduling*; E.1 [**Data structures**]: Graphs and networks

General Terms

Algorithms, Performance, Measurement

1. INTRODUCTION

Recent and future parallel machines for high performance computing offer a very large number of parallel processing units. The immense increase in parallelism of these architectures is caused by multi-core and many-core processors used for HPC systems. Today most parallel machines are equipped with dual or quad-core processors; within a few years a single processor is expected to provide 10s or 100s of execution cores. For the application program, the architectural development towards multi-core systems poses the challenge of providing and programming application codes with a very large degree of potential parallelism. The degree of parallelism within a parallel application code depends on both, 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 the model of hierarchical multiprocessor tasks (M-tasks) for developing application programs for such large parallel systems.

The M-task programming model can be used to structure parallel programs in a flexible way by expressing the available degree of parallelism in the form of M-tasks. This can, for example, be used to combine the benefits of task and data parallelism by using data parallelism within the M-tasks and by expressing the task parallelism as interactions between the 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 describes how the M-tasks cooperate with each other and which dependencies have to be considered for a correct execution. In particular, the coordination structure also identifies which M-tasks can be executed concurrently to each other because there are no dependencies. The M-task structure can be hierarchical and stops with basic M-tasks. These are not further subdivided and are implemented using an SPMD programming style, e.g. by employing MPI or OpenMP, and can run on an arbitrary number of processors or cores. The coordination specification is completely application-specific and is independent from the hardware or interconnection structure of the target platform. This decouples the specification of parallelism from the actual parallel execution on a specific parallel platform and allows a change of the parallel execution without changing the specification of parallelism. For a specific target platform, the M-tasks should be mapped such that the computational work is balanced and the resulting communication overhead is at a minimum. This mapping is done in a separate step which uses the coordination specification as input. The advantage of this approach is that it allows us to increase the available degree of parallelism by defining a suitable M-task structure and to restrict communication within M-tasks to subsets of the available processors. Thus, the communication overhead can be reduced and scalability can be increased.

Many-task computing (MTC)[13] is a new research direction encompassing support for running parallel applications consisting of

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

MTAGS '09 November 16th, 2009, Portland, Oregon, USA

Copyright © 2009 ACM 978-1-60558-714-1/09/11 ...$10.00

many loosely coupled tasks on HPC platforms. In the M-task programming model, the tasks can be tightly coupled using a message passing paradigm, e.g. MPI, for data exchanges between the tasks, or loosely coupled as it is supported for grid-like environments by the TGrid[7] system. The number of tasks in an M-task program usually depends on the characteristics of the parallel algorithm and the communication behavior of the target execution platform. The number of tasks may be very large, e.g. for executing numerical algorithms with a large number of time steps on tightly coupled parallel machines, or quite small, e.g. for executing multiphysics applications on grid environments. Thus, the M-task programming model can be considered as a suggestion for the implementation of many-task applications.

An M-task application program and its coordination specification offer several possibilities for a parallel execution, differing in the order in which the M-tasks of a program are executed and the subsets of processors or cores assigned to each M-task for execution. This is selected in a separate scheduling and mapping step. On different parallel architectures different versions of the M-task program might be the most efficient and scalable ones. To find an optimal M-task program version is an NP-complete problem which is usually solved by scheduling heuristics or approximation algorithms. Most of those existing M-task scheduling algorithms are defined for homogeneous systems or distributed grid-like systems. Several strategies for mapping M-task applications on heterogeneous multi-core platforms have been presented in [5]. In this article, we extend this approach and propose a combined scheduling and mapping algorithm which is aware of the heterogeneity of multi-core systems. 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 algorithm for M-task programs which extends existing scheduling approaches for the use in multi-core systems and reduces the computational effort due to a reduced search space;
- to investigate different mapping strategies for several benchmarks from the area of 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 and thus it is important to provide suitable parallel implementations. We also consider solvers for partial differential equations (PDEs) from the NAS parallel benchmark suite.

The investigations on dual-core and quad-core systems show that the application performance is significantly impacted by both, the selected execution scheme and the applied mapping strategy. Additionally, we show that the performance of ODE solvers with an M-task organization can be improved by exploiting special communication patterns for the interaction of M-tasks based on an orthogonal arrangement of the processes executing the M-tasks.

The rest of the paper is organized as follows. Section 2 gives a short description of the M-task programming model. Section 3 describes scheduling algorithms and mapping strategies for M-tasks. Section 4 presents a detailed experimental evaluation of the mapping strategies for different recent parallel systems. Section 5 discusses related work and Section 6 concludes the paper.

2. M-TASK PROGRAMMING

The M-task programming model is a programming style to code parallel programs in a mixed task and data parallel way using cooperating parallel tasks, which are called M-tasks. An M-task is a piece of parallel program code that can be executed in parallel on a subset (group) of processors and that can cooperate with other M-tasks. Thus, an M-task program is built up from a set of M-tasks cooperating with each other. The coordination between M-tasks can be based on control or data dependencies. M-tasks have a set of input parameters and produce a set of output parameters; both are visible to other M-tasks. A data dependence between M-tasks M_1 and M_2 arises if M_1 produces output data required as an input for M_2. A data dependence may lead to a data re-distribution operation if M-task M_1 provides its output data in a different distribution or on a different set of processors than it is expected by M-task M_2. Depending on the execution platform and the size of the data, such a data re-distribution may be expensive. Therefore, the selection of the processors executing the M-tasks is important and may have a large influence on the total execution time.

Data dependencies or control dependencies emerge from the structure of the application and lead to precedence relations between M-tasks. Due to the precedence relation between M-tasks, M-task programs can be represented by a graph structure $G_M = (V, E)$, where the set of nodes V consists of the M-tasks of a program and edges $e \in E$ connect different M-tasks M_1 and M_2 if there is a precedence relation between M_1 and M_2. A precedence relation can be a data or control dependence from M_1 to M_2. Examples for such graph structures are the macro dataflow graphs used in the PARADIGM compiler[2] or the SP-graphs used in the TwoL model[16]. Figure 1 shows an example for an M-task graph with ten M-tasks.

There may exist several parallel execution orders for a given M-task graph that differ in the scheduling and mapping of M-tasks to a subset of processors or cores of the parallel computing system. Precedence relations restrict the possible execution order of the M-tasks. If M-tasks M_1 and M_2 are connected by a precedence relation the execution of M_1 must have been finished and all required data re-distribution operations must have been carried out before the execution of M_2 can be started. For independent M-tasks a concurrent execution on disjoint subsets of the available processors as well as an execution one after another are possible.

For the parallel execution of an M-task program represented by an M-task graph G_M on a heterogeneous parallel platform there exist several execution schemes differing in

i) the number of cores assigned to each M-task;

ii) the execution order for independent M-tasks, i.e. for M-tasks $M_1, \dots, M_k \in V$ that are not connected by a path in G_M;

iii) the assignment of specific processors (or processor cores) of the execution platform to specific 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

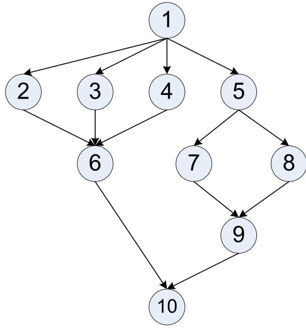

Figure 1: Example for an M-task graph consisting of a set of ten M-tasks $\{1, \ldots, 10\}$.

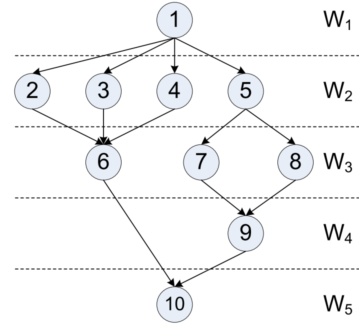

Figure 2: Partitioning of the M-task graph from Figure 1 into five layers $\{W_1, \ldots, W_5\}$.

times may result for the communication within the M-tasks as well as for the re-distribution operations between cooperating M-tasks. An efficient execution scheme can be selected by suitable mapping and scheduling strategies as discussed in the next section.

3. SCHEDULING AND MAPPING

Executing an M-task program on a heterogeneous 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. If the cores assigned to an M-task do not have a shared address space, the data distributions of the input and output parameters also has to be fixed. In the following, we concentrate on the scheduling and mapping decisions.

3.1 Cost model for M-tasks

The scheduling decision has to be based on a cost model for the execution of the M-tasks. The costs T of a single M-task M depend on the computational work $T_{comp}(M, 1)$ 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. This can be expressed by

$$T(M, q, mp) = T_{comp}(M, 1)/q + T_{comm}(M, q, mp)$$

for q cores and mapping pattern mp, assuming a linear speedup for the computational part of the M-task. If the cores do not have a shared address space, there is an internal communication time T_{comm} (M, q, mp) of M which depends on q and the mapping pattern of the cores. In this case, re-distribution costs $T_{Re}(M_1, M_2, q_1, q_2, mp_1, mp_2)$ between cooperating M-tasks M_1 and M_2 may also occur. These depend on the number q_i of cores executing M_i, $i = 1, 2$, and the mapping pattern mp_i used for M_i.

3.2 Scheduling

The scheduling step determines the execution order of the M-tasks within a task graph. In particular, the scheduling step decides for a set of independent M-tasks whether they 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 M-task one after another. It might also be beneficial to use a fixed set of groups of execution cores assigning independent M-tasks to these groups such that each group executes several M-tasks one after another.

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[15]. There are several possibilities to partition an M-task graph into layers. The greatest flexibility for the scheduling decision is achieved by using as few layers as possible. This can be reached by a greedy algorithm that runs top-down over the M-task graph and puts as many nodes as possible in the current layer. Figure 2 illustrates this step for an example graph.

The layers of an M-task graph are scheduled one after another. Within a layer, the M-tasks can be scheduled in an arbitrary way, i.e., 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 M-tasks of the layer. To simplify the algorithm, we make two assumptions: (a) the number of subsets and their size is constant during the execution of the M-tasks of one layer; (b) we use homogeneous symbolic cores to build the subsets. Symbolic cores are later mapped to physical cores of a heterogeneous architecture in the mapping step. The costs for an M-task M on a set of p homogeneous cores is denoted as $T_{symb}(M, p)$ and includes communication costs for a default mapping pattern. 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 step is described later.

The scheduling algorithm considers several reasonable numbers of subsets for a layer. For the actual mapping, we modify a greedy linear-time scheduling algorithm for uniprocessor tasks without dependencies [17] with a proven suboptimality bound of $4/3$. This suboptimality bound does not hold for M-task layers, but it shows good results in practice. Algorithm 1 sketches the key ideas of the scheduling step.

The scheduling algorithm determines for each layer the optimal number of subsets of symbolic cores to be used for the execution of the M-tasks of the layer. The created subsets have equal size. The execution time for a layer W can sometimes be reduced by adjusting the number of cores of the subsets to the accumulated computational work of the M-tasks assigned to the subset G_l which can be defined as

$$T_{seq}(G_l) = \sum_{M_i \in \mathcal{M}_l} T_{comp}(M_i, 1)$$

where $\mathcal{M}_l$ is the set of M-tasks assigned to subset G_l. Assuming malleable M-tasks with linear speedup, the subset size of a layer W can be adapted to the computational work by assigning g_l symbolic

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

begin
  foreach *(layer $W = \{M_1, \dots, M_k\}$)* **do**
    let $P(W)$ be the set of symbolic cores for layer W;
    let $p(W) = |P(W)|$ be the number of symbolic cores;
    $T_{min} = \sum_{i=1}^{k} T_{symb}(M_i, p(W))$;
    foreach *($g \in$ (set of divisors of k))* **do**
      partition $P(W)$ into g subsets $G = \{G_1, \dots, G_g\}$ of size $p_g = p(W)/g$;
      sort $\{M_1, \dots, M_k\}$ such that $T_{symb}(M_1, p_g) \geq \dots \geq T_{symb}(M_k, p_g)$;
      for *($j = 1, \dots, k$)* **do**
        assign M_j to G_l with the smallest accumulated execution time;
      $T_{act}(g) = \max_{1 \leq j \leq g}$ accumulated execution time of G_j;
      if *($T_{act}(g) < T_{min}$)* **then**
        $T_{min} = T_{act}(g)$;
end

cores to subset G_l with

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

For a distributed address space, re-distributions between neighboring layers also can be taken into consideration to further improve the subset selection. Such re-distributions may be necessary for cooperating M-tasks that exchange data and that are executed on different sets of cores. In this case, the subsets should be selected such that the re-distribution costs are minimized. This is often achieved by using subsets of the same size for neighboring layers and assigning M-tasks that are exchanging data to the corresponding subsets.

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 in a tree-structure with cores C as leaves, 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. Figure 3 shows an illustration. For a unique identification of the leaf nodes k of the architecture tree, we use the label $l(k) = n.p.c$ consisting of the node id n, the processor id p and the core id c.

3.4 Mapping

An architecture-aware mapping of an M-task program to a multi-core system is a mapping F from a graph structure G_M to a tree structure A describing the architecture, i.e., $F : G_M \to A$. Since the graph corresponding to an M-task program represents the entire execution and control flow of the program, there has to be a mapping from the graph to the execution platform for each point of the execution time. The graph structure is partitioned into layers

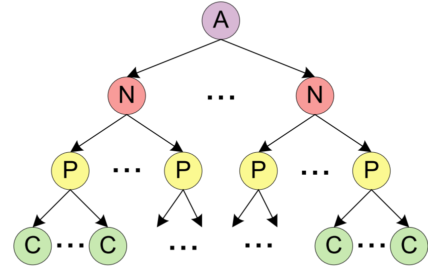

Figure 3: Illustration of a tree representing the architecture of a hierarchical multi-core SMP cluster.

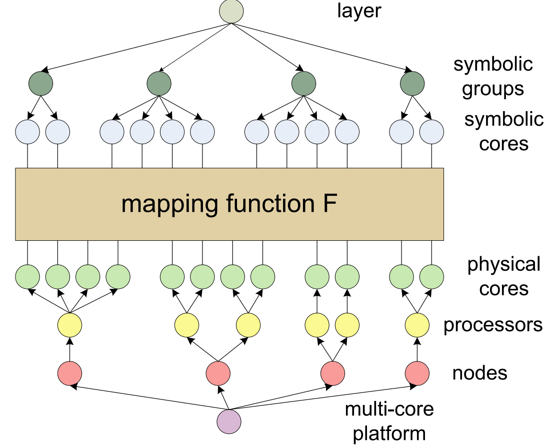

Figure 4: Illustration of the mapping function from symbolic to physical cores.

by the scheduling algorithm such that for each layer a set of groups of symbolic cores is used where each group may execute several M-tasks one after another. The number g_i of symbolic cores of each group G_i is fixed by the scheduling algorithm and the executions performed by a group G_i are independent from the executions performed by G_j, $i \neq j$ with $i, j = 1, \dots, g$ for groups of the same layer. The mapping F is defined for each group partitioning $G = \{G_1, \dots, G_g\}$ of the symbolic cores of each layer W of the input M-task graph. The result of the mapping for the groups of one layer are disjoint sets of physical cores, i.e.

$$F : \{G_1, \dots, G_g\} \to 2^{\mathcal{C}}$$

where $\mathcal{C}$ denotes the set of physical cores; F maps a symbolic group G_i to a physical group $F(G_i) = C_i$ with $F(G_i) \cap F(G_j) = \emptyset$ for $i \neq j$. Moreover $|G_i| = |F(G_i)|$, i.e., each symbolic group is mapped to a physical group of the same size. Figure 4 shows an illustration of the mapping function.

In the following, we propose several mappings mainly differing in the strategies how symbolic cores are mapped to physical cores of the parallel machine. Since the underlying programming model like MPI or OpenMP also influences performance and communication times, the specific choice of the mapping function also has to take this into consideration. For each proposed mapping, we define a sequence of physical cores

$$s_1, s_2, \dots s_{|\mathcal{C}|}.$$

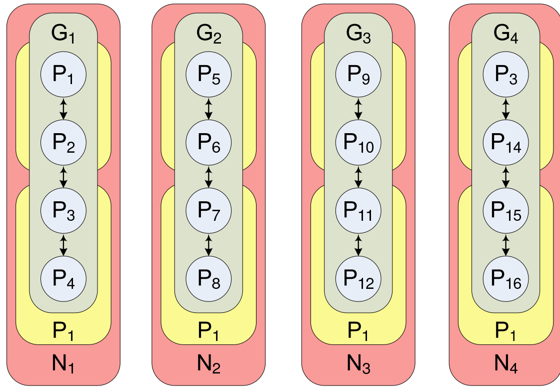

Figure 5: Example for a consecutive mapping of a group partitioning into four symbolic groups $G_1, \ldots, G_4$ each including four symbolic cores on a platform with four nodes consisting of two dual-core processors. The edges symbolize communication within M-tasks.

Each physical core appears exactly once in this sequence. The mapping function F assigns the symbolic cores of a group $G_i, i = 1, \ldots, g$, to consecutive physical cores in this sequence, i.e.

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

Consecutive mapping: For this mapping, symbolic cores are mapped consecutively to physical cores to obtain a node-oriented use of the physical cores. If a group of symbolic cores is larger than the number of physical cores per node of the architecture, several nodes are used such that each node is used only for one group. Otherwise, more than one group may be mapped to one node. This mapping tries to minimize the number of groups that are mapped to each node of the architecture. Figure 5 shows an example. This mapping should be beneficial if communication within nodes is faster than communication between nodes for the specific target architecture and intra M-task communication outweighs inter M-task communication. Furthermore, it enables the use of OpenMP threads for processes mapped to the same node.

In this mapping, the physical cores are ordered such that cores of the same node are adjacent. For example, the sequence of physical cores for a platform with N nodes each consisting of P processors with C cores is given by

$$1.1.1, \ldots 1.1.C, 1.2.1, \ldots, 1.P.C, 2.1.1, \ldots N.P.C.$$

Scattered mapping: For this mapping, the physical cores for a specific group of symbolic cores are selected such that corresponding cores of different nodes are used, see Figure 6 for an illustration. If a group contains less symbolic cores than nodes, one physical core of each node is used for the mapping. This ensures an equal participation of the nodes in the communication performed during the execution of the M-tasks of the group. This mapping should be beneficial if data re-distribution operations between M-tasks outweigh intra M-task communication.

For a platform with N identical nodes each consisting of P processors with C cores the sequence of physical cores is defined as

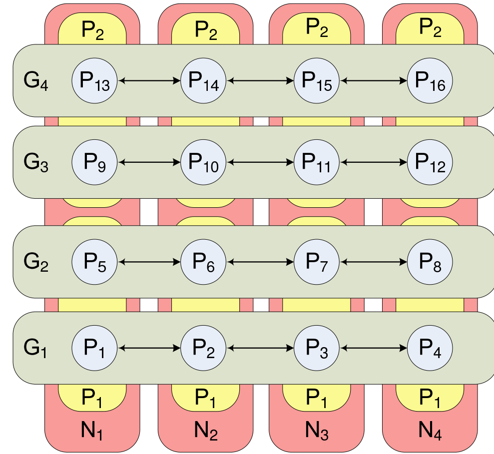

Figure 6: Example for a scattered mapping of symbolic groups $G_1, \ldots, G_4$ with four symbolic cores each on a multi-core platform consisting of four identical nodes each equipped with two dual-core processors.

$$1.1.1, \ldots, N.1.1, 1.1.2, \ldots, N.1.C, 1.2.1, \ldots N.P.C.$$

Mixed mapping: Consecutive and scattered mapping strategies can also be mixed. A parameter d denoting the number of consecutive physical cores of a node used to execute an M-task is used to describe these mappings. If the number of symbolic cores assigned to an M-task is greater d multiple nodes are used for the execution. Choosing $d = 1$ leads to the scattered mapping. For platforms with identical nodes consisting of P processors with C cores each a value of $d = P * C$ corresponds to the consecutive mapping. Therefore, the mixed mapping can be considered the most general case and can be used to adapt to the ratio of intra M-task communication and data re-distribution operations. Figure 7 illustrates this mapping strategy.

4. EXPERIMENTS

In this section, we describe the experimental results obtained by applying the scheduling and mapping algorithms to different application programs.

4.1 Hardware description

For the benchmark tests, a variety of platforms is 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 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 partition containing 128 nodes, each one 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 NUMA-link 4 interconnection network with a bidirectional bandwidth of 6.4 GByte/s per link. The employed MPI library is SGI MPT 1.16 and the Intel Compiler 11.0 is used.

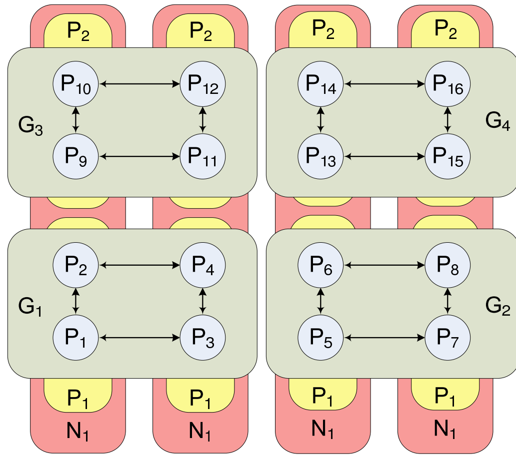

Figure 7: Example for a mixed mapping with $d = 2$ of symbolic groups $G_1, \ldots, G_4$ each consisting of four symbolic cores on a platform with four identical nodes each comprising two dual-core processors.

The **JuRoPA** cluster consists of 2208 nodes with 2 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 Benchmark description

The numerical solution of systems of ordinary differential equations (ODEs) is often based on time-stepping methods that execute a large number of time steps one after another. The computation of a single time step can be based on the evaluation of a fixed number of independent stage vectors and can be represented by an M-task Graph. Coarse grained parallelism is provided between the computations of different stage vectors and fine grained parallelism can be used for a single stage vector by a distributed computation of the components of the ODE system. Examples for explicit solution methods are the Iterated Runge-Kutta(IRK) methods and the Parallel Adams Bashforth (PAB) methods; implicit solvers are Parallel Adams-Moulton(PAM) and Diagonal-Implicitly Iterated Runge-Kutta (DIIRK) methods. The combination of the PAB and PAM methods in a predictor-corrector scheme results in an implicit ODE solver (PABM)[18].

For the benchmarks, three different program versions of these solvers are used. The **data parallel** version computes the stage vectors one after another using all available processors. This corresponds to a partitioning of the symbolic cores of a layer into a single subset in the scheduling algorithm and, thus, leads to many global global communication operations. The standard **task parallel** version computes the stage vectors in parallel on disjoint subsets of the processors. This corresponds to a partitioning of the symbolic cores into the maximum number of subsets. The task parallel version restricts the communication operations required for the stage vector computations to subsets of the processors, but introduces additional global communication to exchange intermediate results between the processor groups. The **task parallel orthogonal** version optimizes the data re-distribution operations by using concurrent multi-broadcast operations on groups of processors and, thus, avoids the global communication of the standard task parallel version.

The ODE systems for the benchmarks include a sparse system that results from the spatial discretization of the 2D Brusselator equation (**BRUSS2D**)[6] and a dense system that arises from a Galerkin approximation of a Schrödinger-Poisson system (**SCHROED**)[14]. The evaluation of a single component requires a fixed number of operations for the sparse system and a linear number of operations for the dense system, respectively.

Another class of applications that can benefit from the M-task programming model are solvers for flow equations that operate 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[19]. The fine grain parallelism within the zones is exploited using shared memory OpenMP programming; the coarse grain parallelism between the zones is realized using message passing with MPI. Therefore, each zone can only be computed within 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 and BT-MZ benchmarks that 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.

4.3 Benchmark results

The measured execution times for the IRK method on the JuRoPA and CHiC clusters for the sparse system are shown in the left column of Figure 8. The mapping stategies are only shown for the fastest program version, which is the task parallel implementation with orthogonal communication in all cases. The CHiC cluster contains 4 cores per node and, thus, a consecutive, a scattered, and a mixed ($d = 2$) mapping are considered. A node of the JuRoPA cluster contains 8 cores and, thus, a mixed ($d = 4$) mapping is also possible. A standard task parallel implementation leads to slower execution times compared to pure data parallelism on both platforms. This can be attributed to the additional communication operations required to exchange intermediate results between processor groups. Utilizing an orthogonal communication scheme leads to much lower runtimes. The lowest execution times are achieved by a consecutive mapping and, on the JuRoPA cluster, by a mixed ($d = 4$) mapping. A scattered mapping is clearly outperformed by the other mappings. These observations are confirmed by the speedup results for the dense system on the JuRoPA cluster that are shown in Figure 8(top right). Compared to the sparse system, the differences between the program versions are smaller because the amount of computation is higher and less time is spent in communication.

Compared to the IRK method, the DIIRK method includes much more communication within the M-tasks that can be restricted to processors groups by a task parallel execution scheme. Therefore, the task parallel and the task parallel orthogonal versions achieve much lower execution times compared to pure data parallelism as it is shown in Figure 8(bottom right) for 512 cores of the CHiC cluster. As for the IRK method, the lowest execution times are

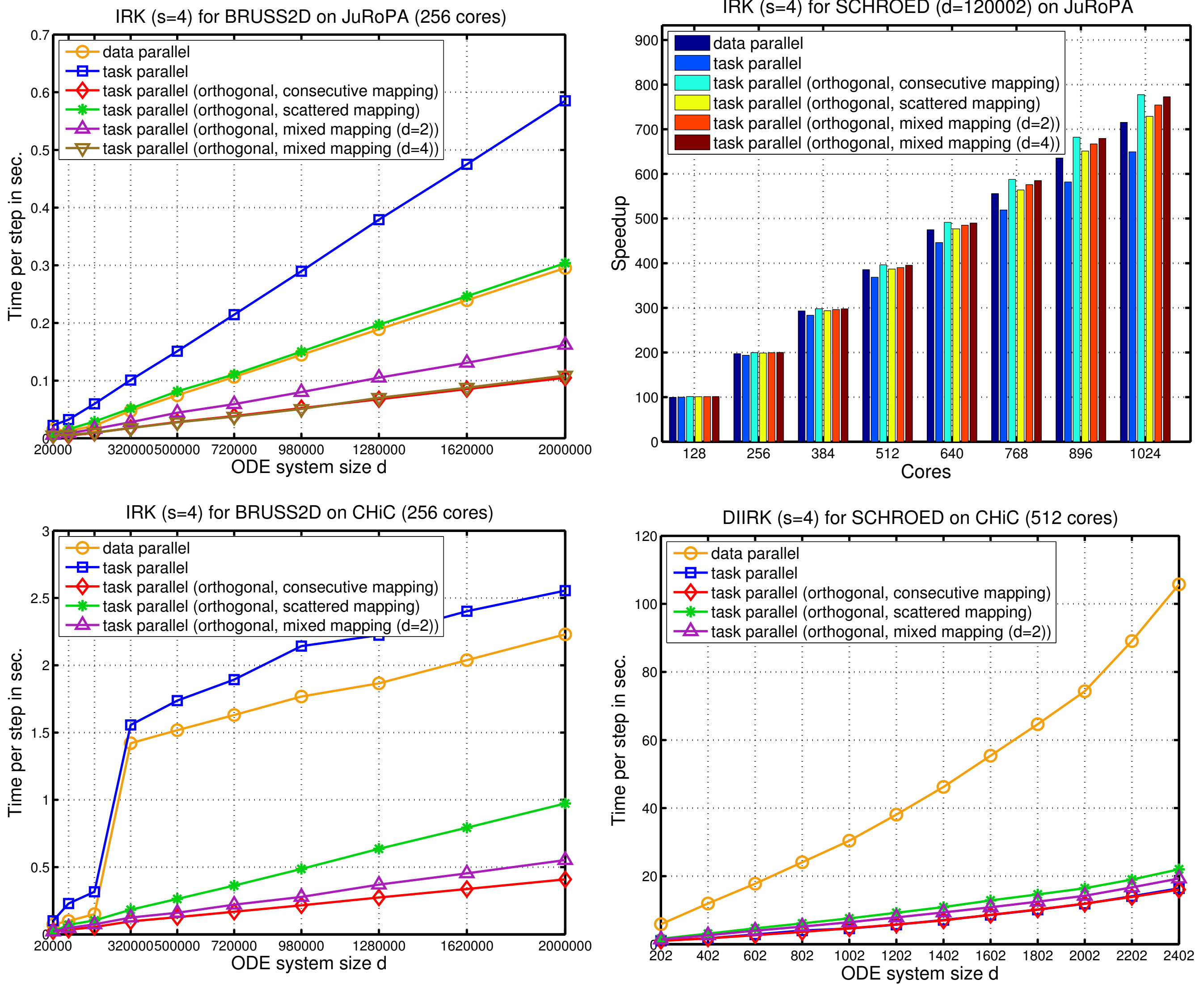

Figure 8: Benchmark results for the IRK and DIIRK (bottom right) methods with $s = 4$ stage vectors on the JuRoPA cluster (top row) and the CHiC cluster (bottom row) using the sparse system (left) and the dense system (right).

achieved by a consecutive mapping.

The top row of Figure 9 shows the measured execution for a single time step of the PAB method. A standard task parallel implementation is not competitive for this method, because the amount of communication within M-tasks is too small and additional global communication is introduced. For the orthogonal program version, communication within M-tasks and communication between processor groups is equally important. Therefore, the mixed mapping strategies with $d = 2$ and $d = 4$ achieve the lowest execution times on the CHiC cluster and on the JuRoPA cluster, respectively.

Compared to the PAB method, the PABM method includes more computation and communication within the M-tasks. Therefore, a placement of the processes executing the same M-task on the same cluster node is desirable. The obtained speedup values for the dense system on the CHiC cluster that are shown in Figure 9(bottom left) confirm this observation. For a high number of processor cores the consecutive mapping of the task parallel orthogonal 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 Figure 9(bottom right) show a similar behaviour, i.e. the consecutive mapping leads to the lowest runtimes and data parallelism is outperformed by all task parallel versions.

The total GFlops per second reported by the SP-MZ benchmark are shown in Figure 10 for the CHiC cluster (top left) and the SGI Altix (top right). The presented results compare different scheduling decisions for a fixed number of cores, i.e. different selections for the number of created symbolic processor groups, for the benchmark classes C and D with 256 and 1024 equal sized zones. The obtained results show that an exploitation of the maximum degree of available task parallelism, i.e. building 1024 groups for class D and 256 groups for class C, does not lead to the highest performance.

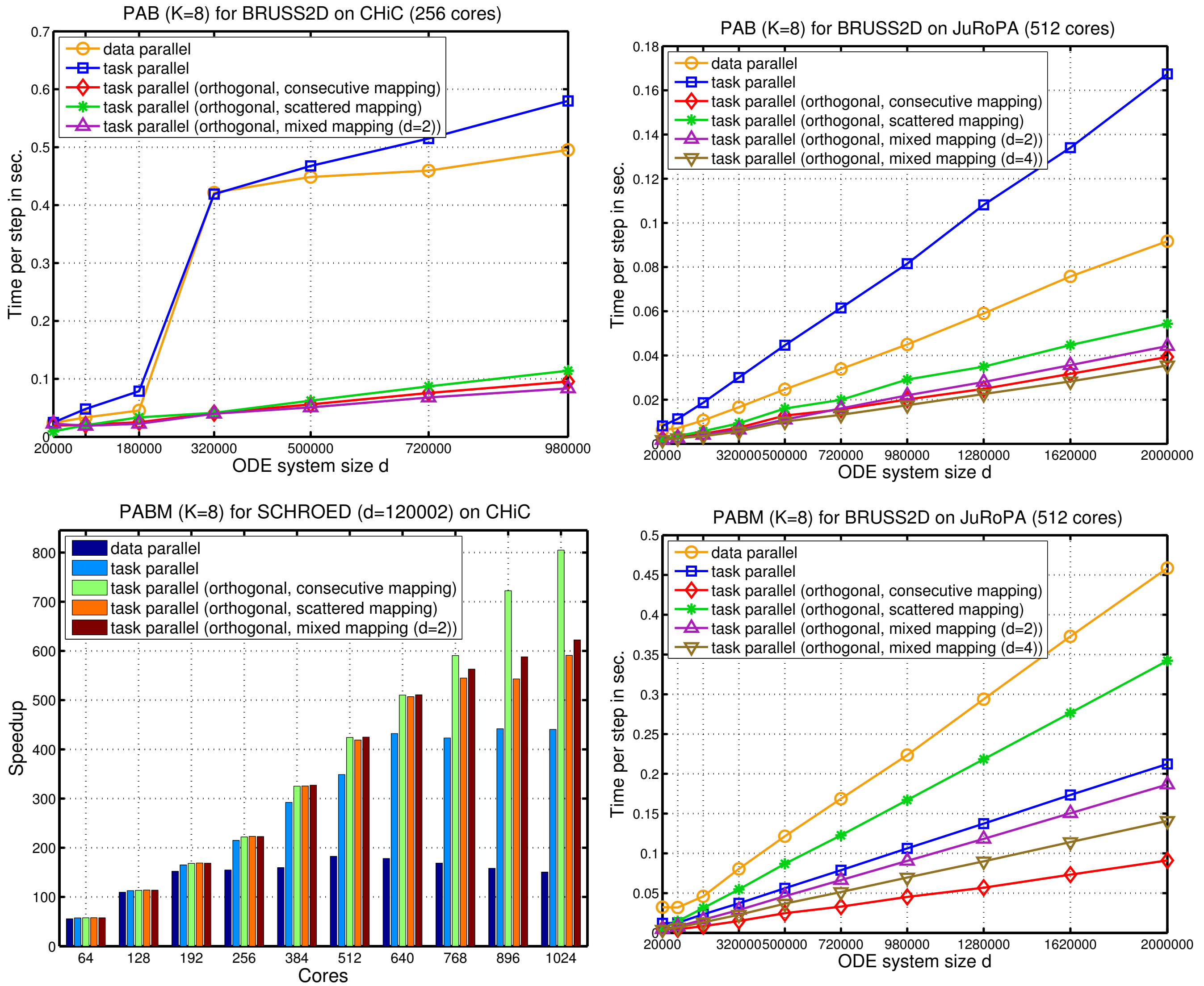

Figure 9: Benchmark results for the PAB (top) and PABM (bottom) methods with $K = 8$ stage vectors on the CHiC cluster (left) and the JuRoPA cluster (right).

On the CHiC cluster, the best execution scheme is 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 highest performance values. 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 many processor cores leading to a high communication and synchronisation overhead within the groups.

The zones of the BT-MZ benchmark incorporate different amounts of computation and, thus, the assignment of M-task to processor groups and load balancing between processor groups becomes an important task. The achieved GFlops/s rates for a varying number of parallel groups are shown in Figure 10 for class C with 256 zones on the CHiC cluster (bottom left) and for class D with 1024 zones on the SGI Altix (bottom right). The highest performance on the CHiC cluster is obtained by the execution schemes with 32 and 128 processor groups. For the SGI Altix, the creation of 32 and 64 processor groups leads to the best results. On both platforms, the scattered mapping outperforms the other mapping strategies. The performance of the execution schemes with many processor groups is degraded by load imbalances introduced by an uneven assignment of the workload.

5. RELATED WORK

Related work comes from M-task scheduling and from mapping parallel applications onto computing ressources. The scheduling of M-task applications on homogeneous target platforms has been investigated by many research groups, see e. g. [20, 12, 3] for a comparison of different scheduling algorithms. The benchmark results presented in this paper show that for multi-core clusters the mapping of processes to cores has a huge influence on the obtained performance. Therefore, the presented combination of scheduling and mapping is a step forward towards a better exploitation of the performance of such platforms.

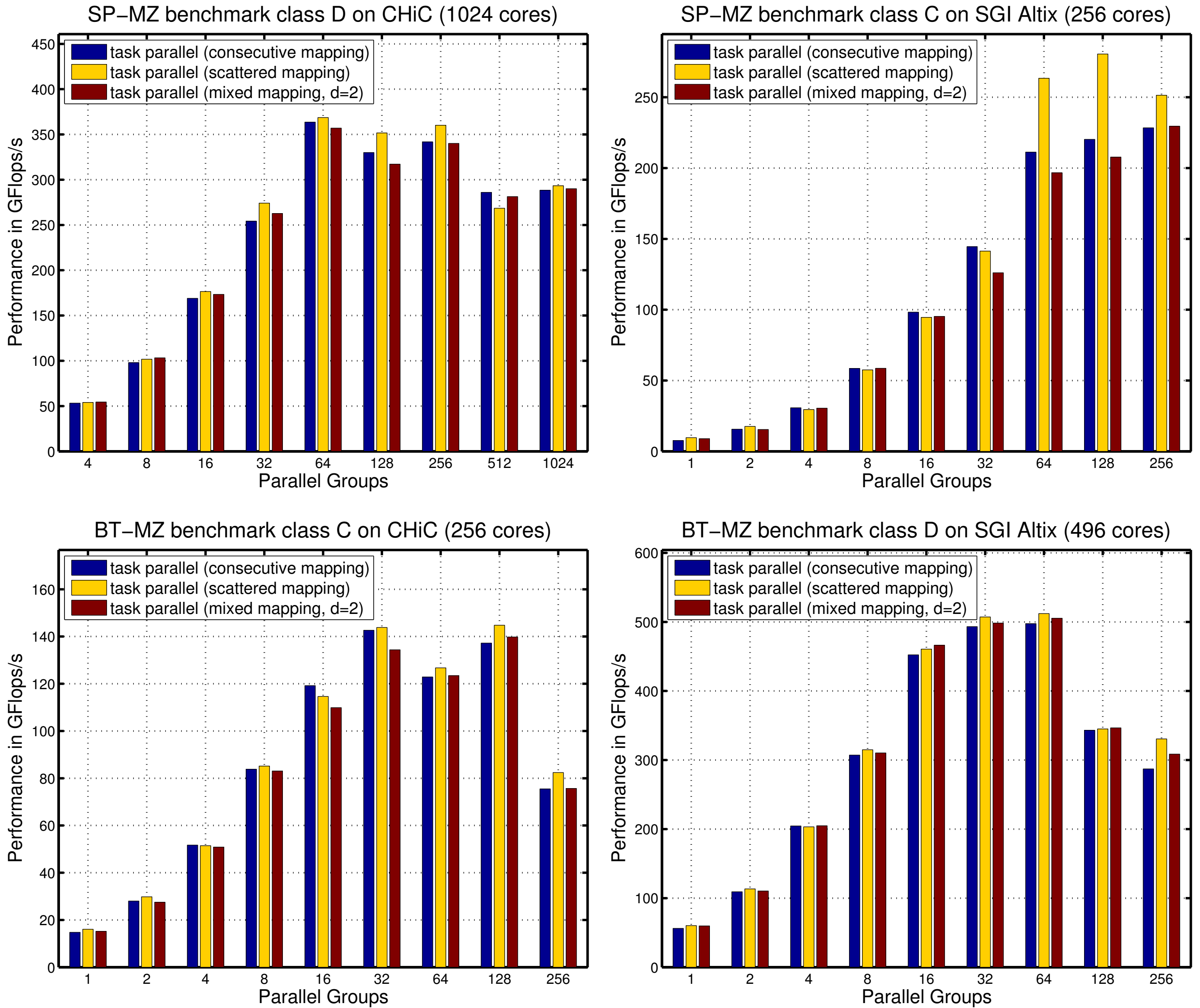

Figure 10: Performance of the SP-MZ (top) and BT-MZ (bottom) benchmarks executed on the CHiC cluster (left) and the SGI Altix (right) using different numbers of parallel processor groups.

An overview of M-task scheduling algorithms for heterogeneous target platforms is given in [10]. These approaches are targeted towards large cluster-of-clusters systems and restrict the execution of an M-task to a single homogeneous sub-cluster. Our benchmark results show that M-tasks have to be executed across multiple nodes of multi-core clusters to obtain high speedups. Therefore, these heterogeneous scheduling algorithms are not suitable for multi-core clusters. In contrast, our approach can determine a suitable task layout for these clusters.

Mapping techniques for parallel applications try to increase application 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. Mixed task and data parallel applications and dependencies between processes are not explicitly taken into account by these approaches.

Multi-core target platforms have been considered in [9]. This approach uses a special graph library for solving the mapping problem. MPIPP[4] is a toolset 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 [11] to map the processes of data parallel applications on target platforms with switch-based

networks.

The mapping of a set of independent tasks each consisting of a fixed number of threads on multi-core platforms has been studied in [8]. The presented 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.

6. CONCLUSION

We have presented a combined scheduling and mapping algorithm for M-task programs on multi-core systems. The scheduling algorithm creates layers of independent M-tasks, partitions the processors into groups of symbolic cores, assigns the M-task of a layer to these groups and adjusts the group 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 M-task to the same cluster node to increase intra M-task communication performance and a scattered mapping that assigns processes of different M-tasks to the same cluster node to improve data exchanges between M-tasks.

Benchmark tests with several large applications from scientific computing show that the M-task approach 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 ordinary differential equations a consecutive placement of the processes of the same M-task onto the same cluster node leads to the best results in most cases. Special orthogonal communication patterns can be used to further increase application performance. The multi-zone benchmarks from the NAS parallel benchmark suite require both, the careful selection of an appropriate number of processor groups and the selection of the suitable mapping strategy. The best performance in the presented measurements was obtained by selecting a medium number of processor groups and using a scattered placement of the processes.

Acknowledgement

We thank the Jülich Supercomputing Centre for providing access to the JuRoPA cluster and the Leibniz Rechenzentrum München for providing access to the SGI Altix.

7. 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 Intl. Parallel and Distributed Processing Symposium (IPDPS'06)*. IEEE, 2006.

[2] P. Banerjee, J. Chandy, M. Gupta, E. Hodge, 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.

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

[4] 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 Supercomp. (ICS'06)*, pages 353–360. ACM, 2006.

[5] J. Dümmler, T. Rauber, and G. Rünger. Mapping Algorithms for Multiprocessor Tasks on Multi-Core Clusters. In *Proc. of the 37th International Conference on Parallel Processing (ICPP 2008)*, pages 141–148. IEEE Computer Society, 2008.

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

[7] S. Hunold, T. Rauber, and G. Rünger. TGrid - Grid Runtime Support for Hierarchically Structured Task-parallel Programs. In *Proc. of the 5th International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Networks (HeteroPar06)*. IEEE, 2006.

[8] Y. Liu, X. Zhang, H. Li, and D. Qian. Allocating Tasks in Multi-core Processor based Parallel Systems. In *2007 IFIP International Conference on Network and Parallel Computing Workshops (NPC'07)*, 2007.

[9] G. Mercier and J. Clet-Ortega. Towards an Efficient Process Placement Policy for MPI Applications in Multicore Environments. In *Proc. of the 16th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 104–115, Berlin, Heidelberg, 2009. Springer-Verlag.

[10] 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 Par. and Distrib. Comp.* IEEE, 2007.

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

[12] 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)*. IEEE, 2001.

[13] I. Raicu, I.T. Foster, and Y. Zhao. Many-Task Computing for Grids and Supercomputers. In *Proc. of the IEEE Workshop on Many-Task Computing on Grids and Supercomputers (MTAGS08)*, pages 1–11. IEEE, 2008.

[14] T. Rauber and G. Rünger. Parallel Solution of a Schrödinger-Poisson System. In *High-Performance Computing and Networking (HPCN)*, volume 919 of *Lecture Notes in Computer Science*, pages 697–702. Springer, 1995.

[15] T. Rauber and G. Rünger. Compiler Support for Task Scheduling in Hierarchical Execution Models. *Journal of Systems Architecture*, 45(6-7):483–503, 1998.

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

[17] S. K. Sahni. Algorithms for Scheduling Independent tasks. *Journal of the ACM*, 23(1):116–127, 1976.

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

[19] 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.

[20] N. Vydyanathan, S. Krishnamoorthy, G. Sabin, U. Catalyurek, T. Kurc, P. Sadayappan, and J. Saltz. An Integrated Approach for processor Allocation and Scheduling of Mixed-Parallel Applications. In *Proc. of the Int. Conf. on Par. Process. (ICPP'06)*. IEEE, 2006.