A Scheduling Toolkit for Multiprocessor-Task Programming with Dependencies

Jörg Dümmler, Raphael Kunis*, and Gudula Rünger

Chemnitz University of Technology
Department of Computer Science
09107 Chemnitz, Germany
{djo, krap, ruenger}@cs.tu-chemnitz.de

Abstract. The performance of many scientific applications for distributed memory platforms can be increased by utilizing multiprocessor-task programming. To obtain the minimum parallel runtime an appropriate schedule that takes the computation and communication performance of the target platform into account is required. However, many tools and environments for multiprocessor-task programming lack the support for an integrated scheduler. This paper presents a scheduling toolkit, which provides this support and integrates popular scheduling algorithms. The implemented scheduling algorithms provide an infrastructure to automatically determine a schedule for multiprocessor-tasks with dependencies represented by a task graph.

1 Introduction

Large applications for distributed memory platforms often show an insufficient scalability for a high number of processors resulting from a large communication overhead especially caused by collective communication operations, like broadcast operations. The increasing popularity of large homogeneous cluster systems and hierarchical cluster-of-cluster systems necessitates a programming model that helps to reduce this overhead. Multiprocessor-task programming can address this problem by splitting an application into a set of multiprocessor-tasks (also called M-tasks or modules), which can be executed in parallel on disjoint processor groups if there are no preventing data dependencies.

The development of M-task applications, which exploit data and task parallelism at the same time, is usually more complex and error-prone compared to the development of pure data or task parallel applications. Therefore a variety of software tools, frameworks and language extensions have been proposed, see e.g. [1–3]. Most of these approaches belong to the class of data parallel languages with support for task parallelism or to the class of task parallel languages with the option to specify data parallelism.

An efficient execution of an M-task application requires a schedule, which defines the execution order and the processor groups for all M-tasks. The schedule

* supported by Deutsche Forschungsgemeinschaft (DFG)

that leads to a minimum runtime strongly depends on the hardware characteristics of the target machine, e.g. the communication performance, and is therefore not portable. Scheduling algorithms that are often based on heuristics or approximation procedures can be used to calculate a schedule depending on the structure of the application and hardware specific parameters. Most of the existing tools used to develop M-task applications require the developer to manually specify a schedule. This increases the required implementation effort and leads to a poor portability.

In this paper we introduce the scheduling toolkit (*STK*) that offers support for scheduling M-task applications and may be used by other tools for M-task programming. *STK* takes a description of the structure of an M-task application in form of a hierarchical directed acyclic graph with annotated cost information and a description of the parallel target platform as an input and produces a feasible M-task schedule that is adopted to the target platform as an output. The cost information in the input may be measured runtimes or runtime predictions in form of symbolic runtime formulas. *STK* contains implementations of popular algorithms for M-task applications with precedence constraints. Furthermore we present a comparison of the implemented algorithms.

This paper is structured as follows. Section 2 explains the programming model used for *STK*. The structure and functionality of *STK* is described in Section 3. The algorithm library of *STK*, which includes all implemented scheduling algorithms, is presented in Section 4, which additionally includes a comparison of the performance of the scheduling algorithms. Section 5 explains the graphical user interface while Section 6 concludes this paper.

2 The M-task Programming Model with Dependencies

An M-task programming model with dependencies was described e.g. in [4] or within the TwoL(Two Level)-system[5]. M-task applications can be modeled by a hierarchical annotated directed acyclic graph (dag) $G = (V, E)$. The node set V in *STK* consists of three different types of nodes: the unique *Start Node* Q that is a predecessor of all other nodes, the unique *Stop Node* R that succeeds all other nodes and a set of regular nodes M that correspond to the execution of an M-task. An M-task $m \in M$ is a parallel program part that can be executed on any group of processors $g_m \in \mathcal{P}(\{1, \ldots, P\}), g_m \neq \emptyset$ of a P-processor homogeneous target platform. The expected execution time is described by the runtime function $T_m : [1, \ldots, P] \rightarrow \mathbb{R}^+$ that is annotated at each M-task $m \in M$.

There are basic M-tasks, whose runtime functions are supplied by the application developer, and complex M-tasks, which consist of other M-tasks and are represented by an M-task dag. The runtime functions for basic M-tasks in *STK* can be specified by either giving measured runtimes or by using symbolic runtime formulas. Symbolic runtime formulas may be derived by adopting a cost model like BSP[6], LogP[7], or by fitting measured runtimes to a function prototype. Depending on the cost model the runtime functions T_m may require further parameters. The runtime functions for complex M-tasks are determined during the

Fig. 1. Example of an M-task application represented by a directed acyclic graph (left) and an appropriate schedule (right). The figure shows the parameter dependencies resulting in a multigraph. The final M-task dag can be derived by combining multiple edges between a pair of nodes into a single edge.

scheduling process by running a scheduling algorithm on the associated internal M-task dag.

Each node $v \in V$ of the M-task dag has a set of input parameters I_v and a set of output parameters O_v, which are data structures used for the communication between M-tasks. Each parameter $a \in \bigcup_{v \in V}(I_v \cup O_v)$ has a data type DT_a, which specifies the size and the memory layout, and a data distribution $DD_a(v)$, which determines which processor owns which part of the data structure. The data distribution may change during program execution and is therefore dependent on the node $v \in V$ that uses the parameter. There is a directed edge $e = (v_1, v_2) \in E$ in the M-task dag if an output parameter of v_1 is an input parameter of v_2 without being modified by another M-task. The edges may be obtained by a dataflow analysis. An edge $e = (v_1, v_2)$ leads to a data re-distribution if the processor group changes, i.e. $g_{v_1} \neq g_{v_2}$, or a parameter requires a different data distribution, i.e. $\exists a \in O_{v_1} \cap I_{v_2}$ with $DD_a(v_1) \neq DD_a(v_2)$. The costs for the re-distribution operation is denoted as $T_{re}(v_1, v_2)$.

There may be multiple implementations of each basic M-task, which accomplish the same task and require the same parameters with the same data types but may use different data distributions or have different runtime estimations. In case of multiple implementations of an M-task $m \in M$ the runtime function T_m and the data distributions $DD_a(m)$ for all $a \in I_m \cup O_m$ additionally depend on the implementation. An example M-task dag is shown in Figure 1(left).

A schedule S assigns each M-task $m \in M$ a processor group g_m and a starting time T_{S_m}, i.e. $S(m) = (g_m, T_{S_m})$. A feasible schedule has to guarantee that all predecessor M-tasks of m have finished their execution and necessary data re-distributions have been carried out before starting the execution of an M-task $m \in M$, i.e. $T_{S_n} + T_n(|g_n|) + T_{re}(n, m) \leq T_{S_m}$ for all $n \in V$ and $(n, m) \in E$. Additionally each processor of the target machine can process one task at any given time index at most, i.e. if $[T_{S_m}, T_{S_m} + T_m(|g_m|)] \cap [T_{S_n}, T_{S_n} + T_n(|g_n|)] \neq \emptyset$ then $g_m \cap g_n = \emptyset$ for all $m, n \in M$. The makespan of the schedule $C_{max}(S)$ is called the time where all output parameters are available at the Stop Node R, i.e.

$$C_{max}(S) = \max_{m \in M}(T_{S_m} + T_m(|g_m|) + T_{re}(m, R)).$$

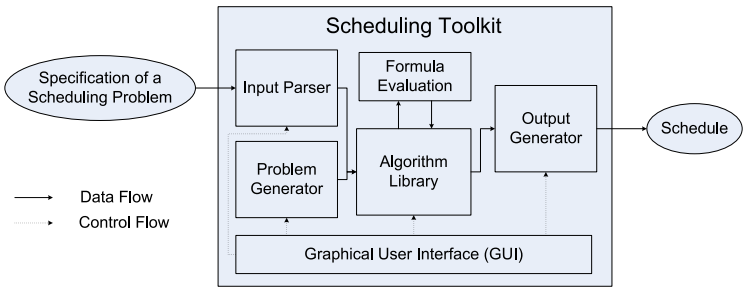

Fig. 2. Dataflow of the scheduling toolkit.

The determination of a feasible schedule with a minimum makespan is called the scheduling problem. Figure 1(right) shows an example of a feasible schedule without data re-distributions.

3 Scheduling Toolkit

This section gives an overview of the structure and functionality of *STK*, which is shown in Figure 2. The workflow of *STK* starts by supplying an input scheduling problem and ends by generating an output schedule. The input scheduling problem can be read from a file by using the *Input Parser* or a synthetic scheduling problem can be created by using the *Problem Generator*.

The input used by *STK* is split into 3 XML-based files, the *SchedulingProblem*, the *ProblemDesc* and the *MachineDesc*. The *SchedulingProblem* input is the main part and describes the structure of an M-task application. This input file consists of a definition of data types and data distribution types and a set of M-task definitions, where each definition contains a parameter list with data types and a set of implementations. Each implementation fixes the data distribution for all parameters of the underlying M-task. Implementations of basic M-tasks furthermore contain a cost information in form of measured runtimes or a symbolic runtime formula. Implementations of complex M-tasks contain an M-task dag. The M-task dag is defined by specifying a set of nodes, where each node refers to an M-task definition in the input file, and a set of edges. Listing 3.1 shows an input description for the example application from Figure 1(left). The definition of data types and data distribution types has been omitted. The root M-task dag representing the whole application is referenced as *MainModule*. As a good schedule for a given application usually depends on the size of the input data or other problem specific values, this information is encapsulated in the *ProblemDesc* input part and may also be changed by using the graphical user interface. In this way, schedules for different problem instances may be created without altering the main input file. An example for such an input file is given in Listing 3.2. Finally all machine dependent information that may be required for symbolic runtime formulas is stored in the *MachineDesc* input part. There

Listing 3.1. Example of a main input file for *STK*.

```
<SchedulingProblem Name="Example">
  <ProblemParam Name="n"/>
  <MachineParam Name="t_op"/><MachineParam Name="t_bc"/>
  <MainModule ModuleRef="2"/>

  <!-- Definition of data types (vector) and data distributions -->

  <Module Name="bm" Id="1">
    <Param Name="in1" Id="1" Type="vector"/>
    <Param Name="in2" Id="1" Type="vector"/>
    <Param Name="out" Id="2" Type="vector"/>
    <Implementation Name="bm_block" Id="1"><BasicModule>
      <Runtime Formula="T_par(p, n, t_op, t_bc)=n*t_op/p+t_bc(p, n)"/>
    </BasicModule></Implementation>
  </Module>

  <Module Name="main" Id="2">
    <Param Name="in1" Id="1"/><Param Name="in2" Id="2"/>
    <Param Name="out" Id="3"/>
    <Implementation Name="main-complex" Id="1"><ComplexModule>
      <!-- Specification of data distributions for all parameters -->
      <StartNode Name="Start" Id="1"/>
      <Node Name="bm#1" Id="2" ModuleRef="1"/>
      <Node Name="bm#2" Id="3" ModuleRef="1"/>
      <Node Name="bm#3" Id="4" ModuleRef="1"/>
      <Node Name="bm#4" Id="5" ModuleRef="1"/>
      <Node Name="bm#5" Id="6" ModuleRef="1"/>
      <StopNode Name="Stop" Id="7"/>
      <Edge Id="1" SourceNodeId="1" SourceParamId="1"
        TargetNodeId="2" TargetParamId="1"/>
      <Edge Id="2" SourceNodeId="1" SourceParamId="2"
        TargetNodeId="2" TargetParamId="2"/>
      <!-- further edge definitions -->
    </ComplexModule></Implementation>
  </Module>
</SchedulingProblem>
```

may be constants, e.g. the time needed to execute an arithmetical operation, and functions, e.g. the time required for a broadcast operation dependent on the size of the data to be transmitted and the number of processors. Listing 3.3 presents an example *MachineDesc* input.

The *Problem Generator* includes several algorithms for creating specific M-task dags, e.g. SP-graphs, in-trees, out-trees, or general dags. SP-graphs [8] are dags that are built according to a recursive definition. An SP-graph is a single node or the series or parallel composition of two SP-graphs. In-trees are a special kind of tree where every node has exactly one outgoing edge meaning all edges are directed towards a single root node. Out-trees are the opposite where every node has exactly one incoming edge and all edges are directed away from a single root node. Furthermore several cost models for representing the computational costs of the nodes of the generated M-task dags are supported. The generated scheduling problems can be used to test the robustness and the performance of scheduling algorithms. The generation of the synthetic M-task dags can be influenced by a number of specifications. These specifications are: type of the dag (SP-graph, in-tree, out-tree or general dag), number of nodes in the dag,

Listing 3.2. Example of a ProblemDesc-file for the extended input format.

```
<ProblemDesc>
  <Constant Name="n" Value="1000"/>
</ProblemDesc>
```

Listing 3.3. Example of a MachineDesc-file for the extended input format.

```
<MachineDesc>
  <Machine Name="CLiC" Processors="12">
    <Constant Name="t_op" Value="0.000000069"/>
    <Function Name="t_bc" Formula=
      "t_bc(p,n)=(0.0383+0.474e-6*log(p))*n"/>
  </Machine>
</MachineDesc>
```

maximum degree of a node (sum of incoming edges and outgoing edges has to be smaller than this value for any node in the dag), a depth ratio to generate flat or deep graphs, a ratio of communication to computation costs, a ratio of the occurrence of more than one implementation for the nodes, and the runtime cost model.

The *Algorithm Library* is the core of *STK* and includes implementations of several scheduling algorithms for M-task programs with dependencies. The scheduling algorithms take an internal representation of a scheduling problem as input and produce an internal representation of a schedule, which can be saved to the output format by the *Output Generator*. The output contains a feasible schedule description for the whole application. This output file of a schedule contains the schedule for the *MainModule* of the input including the length of the schedule, the processor group of the overall scheduling problem, and a list of module calls ordered by their starting time. Each module call is defined by the processors the module is proposed to run on (processor group), the starting time and the finishing time of the module, and the chosen implementation. Parts of the output file for the example input problem are given in Listing 3.4.

STK supports symbolic runtime formulas to represent cost information. This is a very flexible approach as many different cost models can be used to derive these runtime formulas, e.g. BSP, LogP, Amdahl's law, or even unit runtimes. The *Formula Evaluation* subsystem encapsulates all functionality to handle the symbolic runtime formulas.

The *Graphical User Interface*(GUI) enables the user to control all steps in scheduling an M-task dag. Section 5 gives further insight into the features of the GUI. Additionally *STK* may be controlled via a command line interface.

4 Scheduling Algorithm Library

This section gives an overview of the implemented scheduling algorithms. Most of the implemented scheduling algorithms are defined for non-hierarchical M-task

Listing 3.4. Example of an output-file for the given input files.

```
<GlobalSchedule Name="Example">
  <MainSchedule ScheduleRef="1"/>
    <Schedule Id="1" ModuleRef="4" ImplementationRef="1" Length="17.267858">
      <ProcessorMap>
        <Group FirstProc="0" NumProcs="16"/>
        <Machine Name="CLiC"/>
      </ProcessorMap>
      <ModuleCall Name="bm#1" Id="2" StartTime="0.0" EndTime="17.250608"
                  ModuleRef="1" ImplementationRef="1">
        <ProcessorGroup>
          <Group FirstProc="0" NumProcs="6"/>
        </ProcessorGroup>
      </ModuleCall>
      <!-- further module calls sorted by StartTime -->
    </Schedule>
  </MainSchedule>
</GlobalSchedule>
```

dags consisting only of basic M-tasks. Therefore hierarchical input scheduling problems have to be decomposed into a set of non-hierarchical problems first and the resulting schedules of the non-hierarchical schedulers have to be composed into a hierarchical schedule finally. Currently the following algorithms for non-hierarchical M-task dags are contained in our algorithm library:

Allocation-and-Scheduling-based algorithms try to solve the scheduling problem with precedence constraints in a two-step approach introduced in [4] consisting of an allocation step and a scheduling step. Implemented algorithms of this class are *Task-parallel*, *Data-parallel*, *TSAS (Two Step Allocation and Scheduling)*[4], *CPA (Critical Path and Area-based scheduling)*[9], *CPR (Critical Path Reduction)*[10] and a modified version of [11] that we call *MSAA (Modified SP-graph approximation algorithm)*.

Layer-based algorithms are scheduling algorithms, which are based on shrinking and decomposing an M-task dag into layers. These algorithms, which were introduced within the *(TwoL)* system with the *TwoL-Level*[12] and the *TwoL-Tree*[13] algorithm, consist of three phases, the *shrinking phase*, the *layering phase* and the *layer-scheduling phase*. In addition we extended the *Dual-3/2* approximation algorithm[14] for scheduling M-tasks without dependencies by a shrinking and a layering phase for M-tasks with dependencies.

In the following we present a comparison of the implemented scheduling algorithms based on the makespan of the produced schedules. For this purpose we consider the average makespan achieved for 100 different synthetic M-task dags belonging to the class of SP-graphs that were generated by the *Problem Generator* (see Section 3). The generated runtime formulas are based on Amdahl's law. We consider task graphs with 10 to 200 nodes and target platforms with 16 and 256 available processors. The obtained results are presented in Figure 3. Because the results of the *Data-parallel* scheduler are very slow and increase linearly we only show the first values so that the results of the other schedulers can be shown in more detail.

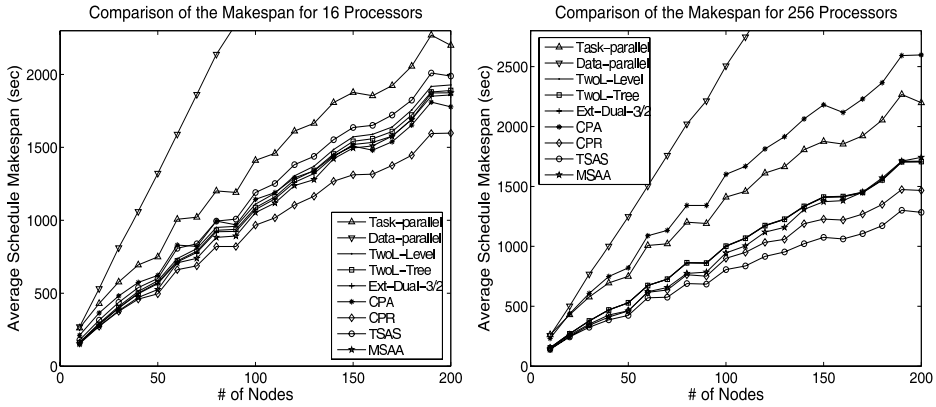

Fig. 3. Comparison of the average makespan of different scheduling algorithms for task graphs with 10 to 200 nodes and platforms with 16 (left) and 256 (right) processors.

The *Data-parallel* and the *Task-parallel* schedulers generate the slowest schedules. We therefore focus on the specialized M-task scheduling algorithms, whose results lie closely together for 16 available processors. For 10 nodes *MSAA* constructs the schedules with the minimum average makespan. For 20 and more nodes *CPR* achieves the best results. The schedules delivered by *TSAS* for 16 available processors have a considerably higher average makespan compared to the best schedulers. Considering the average over all tested numbers of nodes the schedules of *TSAS* have a 22% higher makespan compared to *CPR*.

TSAS, whose performance was rather poor for 16 available processors, achieves the best average results for 256 available processors. For up to 60 nodes *MSAA* produces the second lowest makespans followed by *CPR*. Compared to *TSAS* the schedules constructed by *CPR* have a 12% higher average makespan followed by *MSAA* (18%). *CPA*, whose schedules require 95% more time compared to *TSAS*, is clearly outperformed by the other scheduling algorithms. *Ex-Dual-3/2*, *TwoL-Level* and *TwoL-Tree* exhibit a similar performance by constructing schedules that are about 14% slower compared to *CPR* for 16 processors and about 26% slower compared to *TSAS* for 256 processors.

Considering the obtained schedules for each synthetic M-task dag in isolation it can be seen that the schedule with the minimum makespan is never constructed by the *Task-parallel* or the *Data-parallel* scheduler. *CPA* builds a schedule with minimum makespan in very few cases, it is mostly dominated by *CPR*. *TwoL-Tree* is mostly better than *TwoL-Level* by a small amount. In almost all cases the best schedule is produced by *TSAS*, *TwoL-Level*, *CPR*, *Ext-Dual-3/2*, or *MSAA*. Altogether these results show that a mixed task and data parallel schedule outperforms a pure task or data parallel execution.

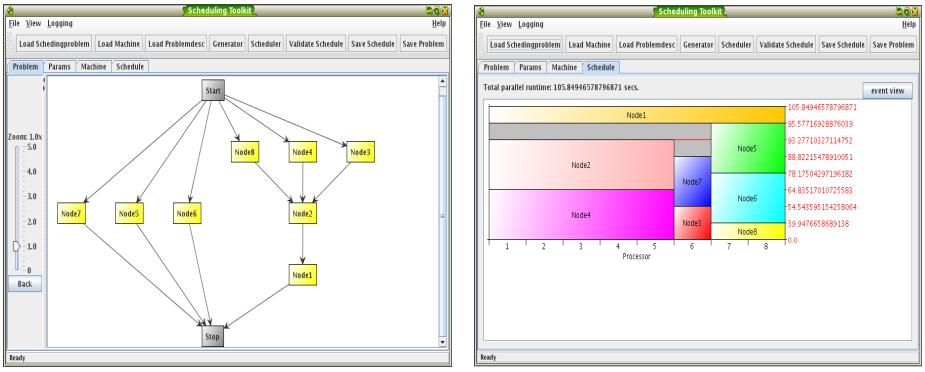

Fig. 4. The scheduling toolkit *STK* with a loaded scheduling problem (left) and the corresponding schedule (right).

5 The Graphical User Interface

The graphical user interface (GUI) of *STK* is shown in Figures 4. The main window provides the functionality to load an M-task dag with annotated runtime information (*Load SchedulingProblem*), to load a specific problem instance (*Load ProblemDesc*), to load a description of a target machine (*Load MachineDesc*), to generate a synthetic M-task dag with synthetic runtime information and a synthetic target machine (*Generator*), to run a scheduling algorithm on the currently loaded problem (*Scheduler*), to validate the obtained schedule (*Validate Schedule*), to save the obtained schedule to an XML file in the output format (*Save Schedule*), and to save the currently loaded problem (*Save Problem*).

The main part of the GUI allows the visualization of a loaded M-task dag presented in Figure 4(left), the editing of problem specific and machine dependent parameters and the visualization of schedules obtained by running one of the included scheduling algorithms shown in Figure 4(right). Furthermore, there are dialogs for the configuration of the scheduling algorithms and for the configuration of the generator for synthetic task graphs.

6 Conclusion and Future Work

In this paper we have introduced the scheduling toolkit *STK*, which offers a scheduling environment for M-task programming with dependencies. The aim of *STK* is to generate a schedule for a specified input scheduling problem, which can be used in isolation or for other tools for the development of M-Task applications. This is necessary because existing tools for developing M-task applications require a manual specification of the schedule. Our solution closes the gap between the specification and the execution of M-task programs by automatically determining good schedules. Through the possibility of including different scheduling algorithms it is possible to adaptively take the applications requirements and hardware details of the target machine into account.

Future steps in the development of *STK* include the support of a wider range of scheduling problems and heterogeneous target platforms. Currently we are working at the integration of *STK* into the TwoL Component System[3].

References

1. Bal, H., Haines, M.: Approaches for integrating task and data parallelism. IEEE Concurrency **6** (1998) 74–84
2. Sips, H., van Reeuwijk, K.: An integrated annotation and compilation framework for task and data parallel programming in java. In: Proc. of 12th Int. Conf. on Par. Comp. (ParCo'03). (2004)
3. Rauber, T., Reilein-Ruß, R., Rünger, G.: On Compiler Support for Mixed Task and Data Parallelism. In: Proc. of 12th Int. Conf. on Par. Comp. (ParCo'03). (2004)
4. Ramaswamy, S., Sapatnekar, S., Banerjee, P.: A framework for exploiting task and data parallelism on distributed memory multicomputers. IEEE Trans. Par. Distrib. Syst. **8** (1997) 1098–1116
5. Rauber, T., Rünger, G.: A Transformation Approach to Derive Efficient Parallel Implementations. IEEE Transactions on Software Engineering **26** (2000) 315–339
6. Valiant, L.G.: A Bridging Model for Parallel Computation. Communications of the ACM **33** (1990) 103–111
7. Culler, D.E., Karp, R., Sahay, A., Schauser, K.E., Santos, E., Subramonian, R., von Eicken, T.: LogP: Towards a Realistic Model of Parallel Computation. In: Proc. of the 4th ACM SIGPLAN Symp. on Principles & Practice of Par. Progr. (PPOPP93), San Diego, CA (1993) 1–12
8. Valdes, J., Tarjan, R.E., Lawler, E.L.: The recognition of series parallel digraphs. Technical report (1979)
9. Radulescu, A., van Gemund, A.: A Low-Cost Approach towards Mixed Task and Data Parallel Scheduling. In: Proc. of the 2001 Int. Conf. on Par. Processing, IEEE Computer Society (2001) 69–76
10. Radulescu, A., Nicolescu, C., van Gemund, A., Jonker, P.: CPR: Mixed Task and Data Parallel Scheduling for Distributed Systems. In: IPDPS '01: Proc. of the 15th Int. Par. & Distr. Processing Symp., IEEE Computer Society (2001) 39
11. Lepere, R., Trystram, D., Woeginger, G.J.: Approximation algorithms for scheduling malleable tasks under precedence constraints. LNCS **2161/2001** (2001)
12. Rauber, T., Rünger, G.: Compiler support for task scheduling in hierarchical execution models. J. Syst. Archit. **45** (1998) 483–503
13. Rauber, T., Rünger, G.: Scheduling of data parallel modules for scientific computing. In: Proc. of the 9th SIAM Conf. on Par. Processing for Scientific Computing (PPSC), San Antonio, Texas, USA (1999)
14. Mounie, G., Rapine, C., Trystram, D.: A $\frac{3}{2}$-Approximation Algorithm for Scheduling Independent Monotonic Malleable Tasks. SIAM Journal on Computing **37** (2007) 401–412