Cluster Computing manuscript No.
(will be inserted by the editor)

SEParAT: Scheduling Support Environment for Parallel Application Task Graphs

Jörg Dümmler · Raphael Kunis · Gudula Rünger

Received: date / Accepted: date

Abstract Programs using parallel tasks can be represented by task graphs so that scheduling algorithms can be used to find an efficient execution order of the parallel tasks. This article proposes a flexible, component-based and extensible scheduling framework called SEParAT that supports the scheduling of a parallel program in multiple ways. The article describes the functionality, and the software architecture of SEParAT. The flexible interfaces enable the cooperation with other programming tools, e.g., tools exploiting a specification of the parallel task structure of an application. The core component of SEParAT is an extensible scheduling algorithm library that provides an infrastructure to determine efficient schedules for task graphs. Homogeneous as well as heterogeneous platforms can be handled. The article also includes detailed experimental results comprising the evaluation of SEParAT as well as the evaluation of a variety of scheduling algorithms.

Keywords Parallel Programming · Scheduling · Task graph · Mixed parallelism · Tool support · Distributed Memory

1 Introduction

Programming with parallel tasks is a suitable programming technique to implement parallel applications consisting of a set of well-defined submodules. Examples are environmental

J. Dümmler
Chemnitz University of Technology, 09107 Chemnitz, Germany
E-mail: djo@cs.tu-chemnitz.de

R. Kunis
Chemnitz University of Technology, 09107 Chemnitz, Germany
E-mail: krap@cs.tu-chemnitz.de

G. Rünger
Chemnitz University of Technology, 09107 Chemnitz, Germany
E-mail: ruenger@cs.tu-chemnitz.de

simulations, aircraft design applications, or large image processing applications. Using parallel tasks, a modular application can be coded as a parallel program with mixed parallelism in which the submodules are implemented as parallel tasks each of which can be executed on one or more processors of a parallel target platform. The processors executing a parallel task start together at a specific start time and execute their parts of the parallel tasks simultaneously until all processors finish their computations by an implicit barrier synchronization. Internally, a parallel task consists of a sequence of parallel instructions comprising computation as well as communication operations. The communication operations may include collective communication operations, which have a high influence on the execution time, especially on distributed memory platforms.

The execution of a parallel task application is based on a schedule that assigns each parallel task a start time and a set of processors of the parallel target platform for execution. A major advantage of the programming with parallel tasks is the flexibility to select an appropriate schedule depending on the number of processors and the communication and computation performance of the target platform. Based on the general parallel task specification, a schedule can be selected such that a machine-specific efficient implementation results. Thus, a portability of efficiency is provided by the programming model.

The schedule of a parallel application can either be specified by hand, which can be error-prone and complex especially for large parallel applications, or can be computed by a scheduling algorithm. The computation of an optimal schedule is a strongly NP hard problem and, thus, scheduling algorithms based on heuristics and approximation algorithms are used. Due to the large variety of parallel applications and scheduling algorithms, it is a challenge for the application programmer to decide which algorithm is suitable for which kind of parallel application to achieve good efficiency results. Thus, a software tool that supports the scheduling of parallel tasks with a variety of algorithms is desirable. Until now, scheduling tools exist only for parallel applications consisting of tasks that are restricted to run on a single processor, e.g., Parallax [23] and CASCH [1]. This article presents the scheduling framework SEParAT (Scheduling Support Environment for Parallel Application Task Graphs), which provides scheduling support for parallel tasks with precedence constraints. An earlier version of SEParAT has been proposed in [13].

SEParAT offers a uniform interface to a variety of static scheduling algorithms for homogeneous as well as for heterogeneous platforms. Heterogeneous platforms especially benefit from the programming with parallel tasks. An example for such platforms are clusters of clusters, i.e., a heterogeneous cluster consisting of multiple homogeneous subclusters. Clusters of clusters usually offer a much higher communication performance within the homogeneous subclusters than between different subclusters. The fine grained data parallel computations inside a parallel task can be mapped to the same subcluster. Communication between different subclusters is then only required between the execution of data dependent parallel tasks occurring at a much coarser granularity.

Two modes of operation are supported by SEParAT: it can be used as an auxiliary tool with a command line interface or as a stand-alone application with a graphical user interface. The command line interface enables the cooperation with other tools that require scheduling decisions. Examples for such tools are parallelizing compilers, or transformation tools that create executable code from a parallel specification, such as the TwoL system [34] or the CM-task compiler [15]. Despite being mainly focused on static scheduling, SEParAT might also be used to provide scheduling support for applications at runtime. The graphical user interface supports the evaluation of existing scheduling algorithms and the development of new scheduling algorithms by a visualization of many internal states, such as intermediate steps of scheduling algorithms. SEParAT also supports the comparison of different schedul-

ing algorithms using benchmarks on a set of synthetic scheduling problems that can be created internally for different user-defined parameters.

The contributions of this article include the definition of the underlying parallel programming model, the description of the interfaces and structure of SEParAT, and a detailed investigation of different kinds of sophisticated scheduling algorithms for parallel tasks with dependencies. The parallel programming model assumed by SEParAT establishes a uniform basis for different approaches of parallel task models and scheduling algorithms. In this model, an application is represented by a hierarchical task graph with annotated cost information. These costs are provided in form of symbolic runtime formulas [6, 18] and, thus, provide the flexibility to adopt a variety of cost models. The programming model supports homogeneous as well as heterogeneous target platforms, such as clusters of clusters. The software architecture of SEParAT is component-based and designed for extensibility. Each of the components can be replaced to adapt SEParAT to specific requirements. For example, the input component can be replaced to support another input format required. The scheduling algorithms are decomposed into phases that are implemented separately as reusable code fragments. This facilitates the implementation of new algorithms, since phases from existing algorithms can be reused. The scheduling algorithm library of SEParAT can be extended with new algorithms using the plugin mechanism provided. An additional feature of SEParAT is its support for several optimizations for existing scheduling algorithms to improve the behavior of the algorithms in terms of the resulting schedules.

This article is organized as follows. The programming model with parallel tasks is defined in Sect. 2. Section 3 gives an overview on scheduling support tools and describes the main functionality of SEParAT. Section 4 outlines the structure of SEParAT and presents the main components and interfaces. Section 5 discusses experimental results. Section 6 concludes the article.

2 Programming model for parallel tasks with dependence constraints

The parallel programming model of SEParAT assumes an application task graph consisting of parallel tasks and their dependencies, a parallel target platform with distributed memory, and a cost model, as described in the following.

Application task graph The structure of a parallel application is represented by a hierarchical annotated directed acyclic graph $G = (V, E)$. The node set V of G consists of a unique entry node q that precedes all other nodes of the graph, a unique exit node r that succeeds all other nodes of the graph, and a set $M = V \setminus \{q, r\}$ of inner nodes. The entry node and the exit node are not associated with computations and represent the input and the output of the application, respectively.

The set M of inner nodes represents the parallel tasks of the application where each parallel task is assumed to be executable on an arbitrary number of processors of a distributed memory platform. Each parallel task $v \in V$ defines a set of input parameters I_v that have to be available for the execution of v, and a set of output parameters O_v that are available when v finishes its execution. Each parameter $a \in I_v \cup O_v$ has a fixed data type that specifies its size and memory layout, and a data distribution type that defines how the individual elements of a are distributed over the set of processors executing the parallel task v.

A parallel task can either be basic or complex. Basic parallel tasks are implemented directly by the application developer, e.g., by using a message passing library, such as MPI. Thus, their internal structure is not visible to SEParAT. For each basic parallel task, there

may exist multiple implementation variants (also called module variants) that share the same interface consisting of input and output parameters and their associated data types. Different module variants of the same parallel task might define different data distribution types for the parameters or might have a different execution time. Complex parallel tasks are composed of other parallel tasks and are represented by an application task graph leading to a hierarchical structure.

The set E of edges of the application task graph represents control and data dependencies between the parallel tasks that have to be taken into account for a correct execution of the application. A control dependency defines that a node of the task graph needs to be finished before another node of the task graph can start. A data dependency edge $e = (v_1, v_2) \in E$ between parallel tasks v_1 and v_2 denotes that v_1 produces one or more output parameters that are required as an input for v_2, i.e., $O_{v_1} \cap I_{v_2} \neq \emptyset$. Data dependencies might lead to data re-distribution operations at runtime of the application, if v_1 and v_2 are executed on different sets of processors, or if v_1 produces its output in a different data distribution than it is expected by v_2 for its input.

Target platform The parallel target platform is a distributed memory machine, which can be either homogeneous, e.g., a single cluster, or heterogeneous, e.g., a cluster consisting of multiple homogeneous subclusters. A homogeneous parallel platform consists of P identical processors that are interconnected by a homogeneous communication network so that the communication performance is identical between each pair of processors. The average time to execute an arithmetic operation is denoted as t_C in the following. The communication performance of the interconnection network is characterized by its startup time t_S and its byte transfer time t_B.

In this article, a heterogeneous parallel platform is composed of c homogeneous subclusters $C^1, \ldots, C^c$ that are connected by a heterogeneous network. Cluster C^i consists of P^i identical processors and its computation and communication performance is captured by the average processing time t_C^i of one operation, the network startup time t_S^i and the network byte transfer time t_B^i of the network, $i = 1, \ldots, c$. The interconnection between subclusters C_i and C_j is defined by the network startup time $t_{S1}^{(i,j)}$ on C^i, the startup time $t_{S2}^{(i,j)}$ on C^j and the byte transfer time $t_B^{(i,j)}$, $1 \leq i, j \leq c$, $i \neq j$.

Cost model The nodes of the application task graph are annotated with a cost function that provides an estimation of the execution time of the corresponding parallel task depending on the set of executing processors. For homogeneous target platforms, these costs are described by a function

$$T_{par} : V \times [1, \ldots, P] \rightarrow \mathbb{R}^+$$

where $T_{par}(v, p)$ denotes the execution time of parallel task $v \in V$ executed on p processors. The cost functions for basic parallel tasks are symbolic formulas in closed form that are specified by the user. These formulas can be derived by fitting measured execution times to an appropriate function prototype, or by adopting a cost model like BSP [17], LogP [10], or LogGP [2], see also [6, 18] for more information on obtaining such runtime formulas. Depending on the approach taken, the function T_{par} may require further parameters. The cost functions of complex parallel tasks are derived by SEParAT based on the corresponding task graph given for the application.

For heterogeneous target platforms, SEParAT restricts the execution of one parallel task to subsets of processors of the same homogeneous subcluster. This assumption is reasonable,

since the interconnection network within a subcluster is usually much faster compared to the interconnection between different subclusters. The costs for executing a parallel task on a heterogeneous platform depend on the executing subcluster and the number of processors used for the execution. This is captured by an annotation of the task graph such that each node is annotated with a set of functions

$$T^i_{par} : V \times [1,\ldots,P_i] \rightarrow \mathbb{R}^+$$

where $T^i_{par}(v,p)$ denotes the execution time of parallel task $v \in V$ executed on p processors of subcluster C^i, $i = 1,\ldots,c$.

The edges of the application task graph are associated with communication costs that result from data re-distribution operations. These costs depend on the amount of data to be transferred from the source task node to the target task node and on the sets of processors used to execute the source and target parallel task, respectively. The amount of data is zero for control dependency edges and is usually larger than zero for data dependency edges. For homogeneous platforms these costs are captured by a function

$$T_{Re} : E \times [1,\ldots,P] \times [1,\ldots,P] \rightarrow \mathbb{R}^+$$

where $T_{Re}((v_1,v_2),p_1,p_2)$ denotes the data re-distribution costs between source parallel task $v_1 \in V$ executed on p_1 processors and target parallel task $v_2 \in V$ executed on p_2 processors. For heterogeneous platforms, the specific subclusters used to execute the source and target parallel tasks need to be taken into account. Thus, the data re-distribution costs are captured by a set of functions

$$T^{(i,j)}_{Re} : E \times [1,\ldots,P_i] \times [1,\ldots,P_j] \rightarrow \mathbb{R}^+$$

where $T^{(i,j)}_{Re}((v_1,v_2),p_1,p_2)$ denotes the communication cost between parallel task $v_1 \in V$ executed on p_1 processors of subcluster C^i and parallel task $v_2 \in V$ executed on p_2 processors of subcluster C^j, $i,j = 1,\ldots,c$.

Scheduling The execution of a parallel application consisting of parallel tasks is based on a schedule S that assigns each node $v \in V$ of the application task graph a subset PG_v of processors (also called processor group of v) and a starting point in time ST_v. This is denoted as $S(v) = (PG_v, ST_v)$. In case of a heterogeneous platform the processor group also encodes the subcluster on which a parallel task is executed, e.g., by using a consecutive numbering of all processors of the platform. The finish time FT_v of a node $v \in V$ is the point in time when the corresponding parallel task terminates its execution. For homogeneous platforms it is computed by $FT_v = ST_v + T_{par}(v,|PG_v|)$, and for heterogeneous platforms by $FT_v = ST_v + T^i_{par}(v,|PG_v|)$ where v is executed on subcluster C^i.

A schedule is called feasible if it fulfills the following constraints. First, a feasible schedule has to guarantee for each parallel task v that all predecessors of v have finished their execution and that all necessary data re-distribution operations have been finalized before the execution of v is started. For two parallel tasks $v,u \in V$ that are connected by an edge $(v,u) \in E$ this constraint can be captured by the condition $ST_u \geq FT_v + T_{Re}(v,u)$ (homogeneous platforms), or $ST_u \geq FT_v + T^{(i,j)}_{Re}(v,u)$ (heterogeneous platforms assuming v is executed on subcluster C^i and u is executed on subcluster C^j). Additionally, a feasible schedule has to define disjoint subsets of processors for parallel tasks with overlapping execution time intervals, i.e., if $[ST_v,FT_v] \cap [ST_u,FT_u] \neq \emptyset$ then $PG_v \cap PG_u = \emptyset$ for all $v,u \in V$. This condition ensures that at any point in time each processor executes one parallel task at most.

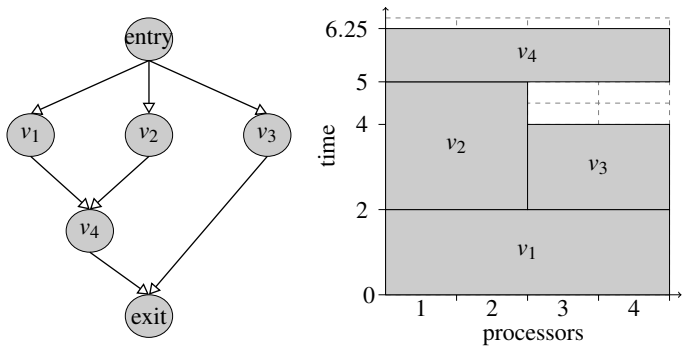

Fig. 1 Task graph of a parallel application (left) and a feasible schedule represented as a Gantt-chart (right). Edges are considered to be control dependencies and have a communication cost zero.

The *makespan* $C_{max}(S)$ of a schedule S is defined as the finish time of the exit node $r \in V$, i.e., $C_{max}(S) = FT_r$. This corresponds to the point in time when all parallel tasks have been executed and the data re-distribution operations for the output parameters have been performed. The determination of a feasible schedule with a minimum makespan is called scheduling problem for parallel tasks with dependencies. This scheduling problem is strongly NP hard even for the special case of precedence constraints in the form of chains [11].

An example for a task graph with a corresponding feasible schedule represented as a Gantt chart is shown in Fig. 1. In this schedule, node v_1 is scheduled to processor group $\{1,2,3,4\}$ starting at time 0. Afterwards, the nodes v_2 and v_3 are executed concurrently on disjoint processor groups $\{1,2\}$ and $\{3,4\}$. The successor node v_4 of the nodes v_1 and v_2 is scheduled to processor group $\{1,2,3,4\}$; the execution starts at the finish time $FT_{v_2} = 5$ of node v_2. The makespan of the schedule corresponds to the finish time $FT_{v_4} = 6.25$ of node v_4.

Scheduling algorithms Different approaches have been proposed for scheduling parallel tasks with dependencies. Most algorithms for homogeneous target platforms belong to one of the following three categories. *Allocation-and-scheduling-based algorithms* [12] consist of two phases: an allocation phase that determines the number of processors for each parallel task and a scheduling phase that maps the parallel tasks to the target platform and defines the start time for each parallel task. Examples for such algorithms are Critical Path and Area-based Scheduling (CPA) [30], Critical Path Reduction (CPR) [29], Two Step Allocation and Scheduling (TSAS) [31], and the approximation algorithms proposed in [21] and [22].
Layer-based algorithms [14] consist of four phases that (i) simplify the task graph, (ii) determine sets of independent tasks of the task graph (also called layers), (iii) schedule the independent tasks of each layer, and (iv) combine the layer schedules to the schedule of the entire task graph. Examples for layer-based algorithms are TwoL-Level [32], TwoL-Tree [33]. Using the layer-based approach, scheduling algorithms for independent parallel tasks can be extended to support dependencies, since these algorithms can be adopted in phase (iii). Examples for scheduling algorithms for sets of independent parallel tasks are Approx-2 [24], Approx-$\sqrt{3}$ [25], and Approx-$\frac{3}{2}$ [26].
Configuration-based algorithms use predefined configurations of processors to schedule the task graph in a single step. The scheduling algorithm OneStep [7] belongs to this category.

Heterogeneous algorithms are designed for heterogeneous cluster of clusters platforms. Examples are Heterogeneous Critical Path and Area-based (H-CPA) [27] and the two scheduling algorithms proposed in [9] called Mixed-Heterogeneous Earliest Finish Time (M-HEFT1 and M-HEFT2). H-CPA extends CPA with support for heterogeneous target platforms. The algorithms M-HEFT1 and M-HEFT2 are an adaptation of the algorithm Heterogeneous Earliest Finish Time (HEFT) [37] with concepts of configuration-based algorithms [7].

3 Related work and usage of SEParAT

This section discusses related work on parallel programming tools with scheduling support and includes an overview of the usage scenarios of SEParAT showing the new functionalities.

3.1 Related work

Although many static scheduling algorithms for parallel applications with precedence constraints have been proposed in the literature, e.g., [5, 22, 27, 29–31], very few scheduling frameworks exist. The existing frameworks mainly support a single algorithm or a very small set of algorithms. To the best of our knowledge there is no framework that supports the scheduling of parallel tasks with is based on a unifying programming model and includes a variety of scheduling algorithms for homogeneous and heterogeneous platforms.

Parallax [23] is a scheduling tool that incorporates multiple scheduling heuristics. It enables the comparison of the heuristics for real-world applications and parallel machines. The input program must be given as a task graph, as in our scheduling framework. The main difference is that Parallax supports only single processor tasks, i.e., tasks that are executed on a single processor of the target platform. Another scheduling tool for single processor tasks is CASCH [1]. It is an integrated programming environment that reads a sequential program (C program with annotations) and generates parallel code based on the annotations. The sequential program is internally transformed into a task graph. Similar to SEParAT and Parallax, it has a library of scheduling algorithms.

HyperTool [39] and PYRROS [40] are tools that automatically produce parallel code from sequential programs. They use a scheduler internally. None of them considers parallel applications with parallel tasks. Furthermore, these tools do not allow the adaption of the scheduling algorithm to the needs of the application, as it is our goal. Also, they do not provide a visualization of the produced schedules to allow a deeper analysis and comparison by the programmer.

PARADIGM [4] and the Spar/Java-Compiler [35] are examples for parallelizing compilers that consider parallel tasks with dependencies. The input of a parallelizing compiler is a sequential program and the output is an automatically parallelized version of the sequential program. However, parallelizing compilers typically provide either no static scheduling support or restrict the scheduling to the use of one built-in algorithm. In contrast, SEParAT provides multiple algorithms and, thus, the most appropriate algorithm for a given scheduling problem can be selected.

Another related field is Grid scheduling. A Grid can be seen as a special type of cluster, that is composed of many independent loosely coupled computers or cluster systems that are connected by a heterogeneous network. Typically, a grid middleware includes a

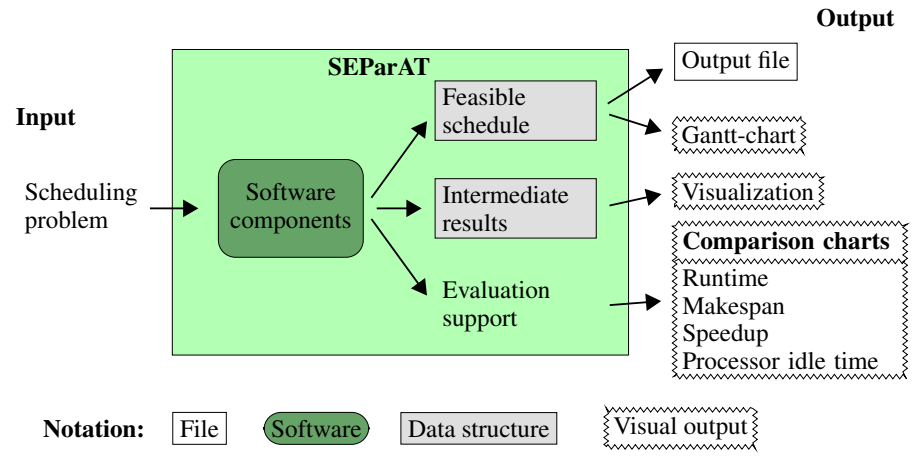

Fig. 2 Overview of SEParAT and its interfaces. SEParAT processes a given scheduling problem and computes a feasible schedule using an appropriate scheduling algorithm. Various further steps are possible.

scheduling component needed to decide which program should be executed on which part of the Grid. Examples are Condor [36], Grasp [20], and ProActive [3]. Most of the scheduling techniques on Grid systems refer to job scheduling where a job is a whole application, independent of other jobs, and coarse-grained. In contrast, the programming model of the proposed scheduling framework SEParAT is based on task scheduling for a single application where the tasks have dependencies and the application is modeled as a task graph, supporting coarse-grained as well as fine-grained parallelism. There are additional circumstances in Grid systems that may influence scheduling decisions, such as service level agreements, accounting and billing, and security [16]. These services are irrelevant for SEParAT, since it considers clusters and clusters of clusters.

The prediction of the runtime and performance of parallel tasks plays an important role for scheduling, since the scheduling algorithms compute a schedule based on this cost information. There are different approaches of performance modeling, which include analytical modeling, simulation modeling, and measurement. In most cases a combination of the three approaches is performed. Examples for the analytical modeling used in SEParAT are given in [18, 38].

3.2 Usage scenarios of SEParAT

SEParAT is flexible, extensible and provides a set of software components with well-defined interfaces to allow the adaptation to a large variety of needs of parallel application programmers. A coarse overview of SEParAT and its interfaces is depicted in Fig. 2. The flexibility is illustrated by a variety of usage scenarios, see Fig. 3 for an overview. The usage scenarios can be divided into three main categories: Scheduling, Evaluation support, and Scheduling algorithm development. In the following, these usage scenarios are described in detail.

Scheduling usage scenarios deal with the computation of a feasible schedule for a given scheduling problem provided either by a parallel programming tool or by an application developer.

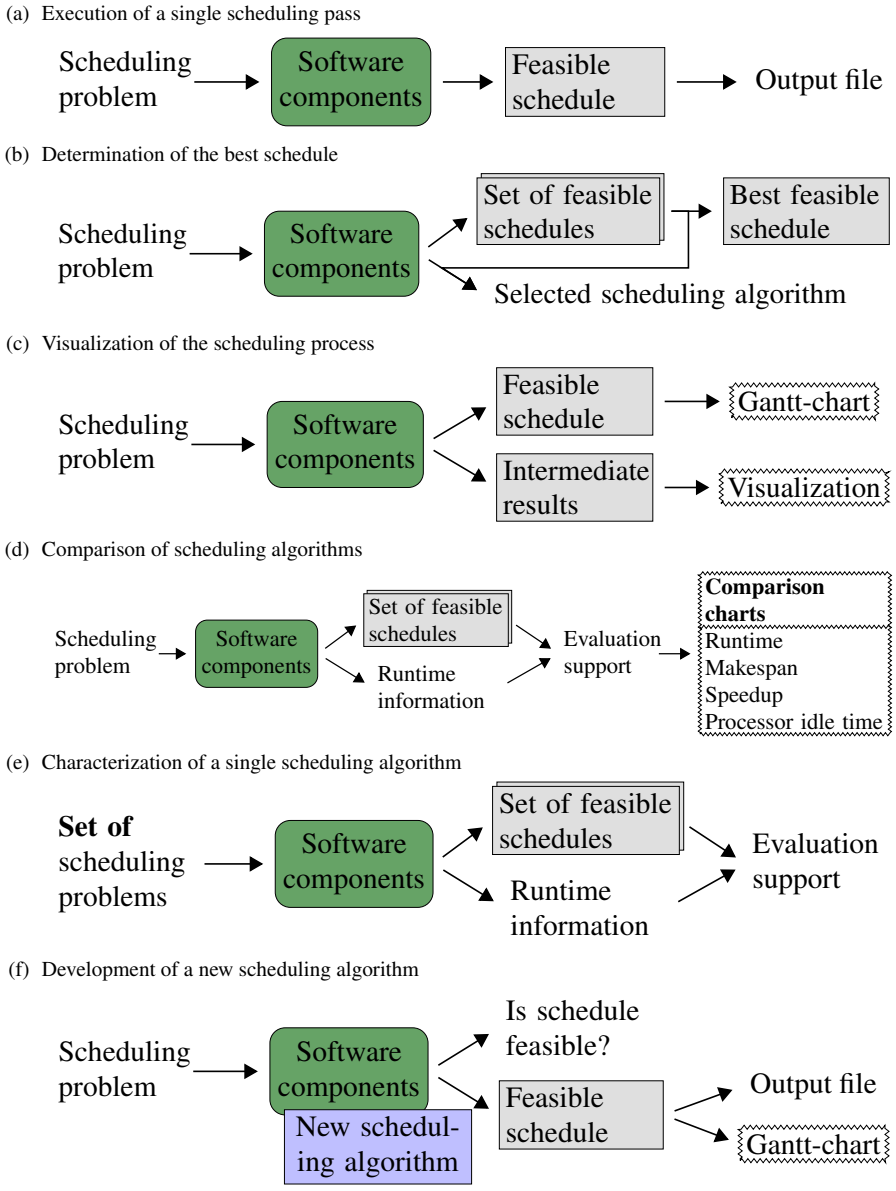

Fig. 3 Illustration of possible usage scenarios of SEParAT.

In usage scenario (a) from Fig. 3, the parallel application programmer already knows the scheduling algorithm that fits best his needs and provides this information as part of the input. In usage scenario (b) from Fig. 3, the appropriate scheduling algorithm is selected by SEParAT. In this case, the user can provide a specific criterion, e.g., the minimization of the schedule makespan. For the determination of the best schedule, SEParAT evaluates the structure of the given parallel application task graph and applies a single scheduling

algorithm or a small selection of scheduling algorithms to the scheduling problem. The best solution found as well as the scheduling algorithm used are delivered as an output.

Evaluation support usage scenarios deal with the comparison of scheduling algorithms for different kinds of input scheduling problems.

The usage scenario Fig. 3 (c) visualizes the layout of the schedules computed by different scheduling algorithms and the available intermediate results. Intermediate results help to get further insights into scheduling algorithms and to identify possible aspects for improvement. Intermediate results may include the number of processors computed for each node of the task graph before assigning the final processors, or the sets of independent nodes that are computed when executing a layer-based scheduling algorithm, see [14].

Usage scenario (d) from Fig. 3 compares a set of scheduling algorithms by applying them to a single scheduling problem. The resulting charts enable the analysis of the scheduling performance when changing the parameters of a scheduling problem or the scheduling algorithm and help developers to decide, which scheduling algorithm to choose for which task graph-target platform combination. SEParAT supports the following different comparison criteria: the execution time of a scheduling algorithm, the makespan of the computed schedule, the speedup of a parallel execution over a sequential execution on one processor, and the processor idle time of the computed schedule.

The evaluation support usage scenario Fig. 3 (e) applies a single scheduling algorithm to a set of scheduling problems. The aim is to compare different characteristics, e.g., the execution time of the scheduling algorithm, the makespans of the computed schedules, and the processor utilization of the computed schedules for large sets of scheduling problems. The set of appropriate scheduling problems is generated by SEParAT for different customizable task graph characteristics.

Scheduling algorithm development usage scenarios enable users to implement new scheduling algorithms. The integration of new scheduling algorithms is an integral part of SEParAT. A user-provided scheduling algorithm can then be used and evaluated like the pre-implemented scheduling algorithms of SEParAT. An illustration of this usage scenario is given in Fig. 3 (f).

4 Functionality of SEParAT

SEParAT exhibits a component-based structure where the components implement different functionalities, see Fig. 4 for an overview of the structure and the steps performed in a single scheduling pass. In the following, the functionality is described in detail. The specification mechanisms are described in Subsect. 4.1, the internal processing and transformation is covered in Subsect. 4.2, and the user interface is described in Subsect. 4.3.

4.1 Specification mechanisms

The main interfaces of SEParAT in terms of input and output structures were chosen according to the definition of the parallel programming model in Sect. 2. The specification of an input scheduling problem is described in Subsect. 4.1.1 and the output produced by SEParAT is explained in Subsect. 4.1.2.

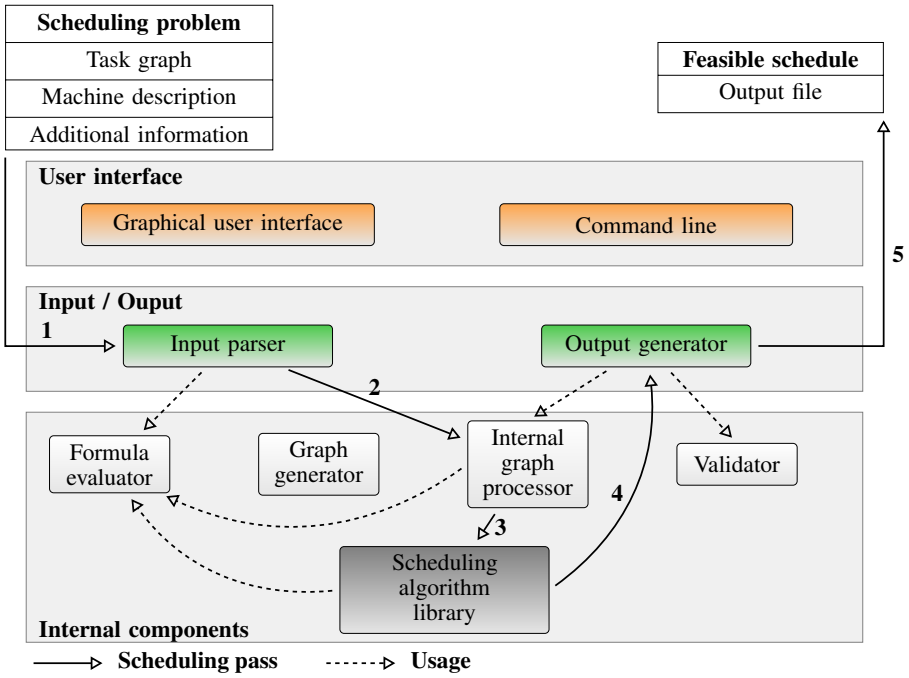

Fig. 4 Functionality and organization of SEParAT. A scheduling pass consists of the following steps: **1** creation of the internal structures from the input specification, **2** transformation and simplification of the internal structures, **3** application of a scheduling algorithm, **4** postprocessing of the computed schedule, and **5** generation of the output files. The scheduling pass is managed via the User interface.

4.1.1 Scheduling problem specification

The specification of a scheduling problem for SEParAT consists of three parts: the parallel application task graph, the properties of the target platform, and (optional) additional information that define parameters for a specific problem instance, e.g., the input data size of the application to be scheduled. The independence of the task graph from the problem size and from the number of processors is an important property for task graph models [8]. The input in form of a task graph decouples SEParAT from specific programming languages and, thus, enables SEParAT to cooperate with a variety of other tools.

The application task graph is specified in an input file consisting of four parts: external parameter declarations, data type definitions, definitions of the parallel tasks, and the specification of a distinguished complex parallel task that represents the entire application. An example is shown in Fig. 5. The external parameters declared in lines 1-3 are provided either in the machine description or in the additional information input files. The data types of the input and output parameters of the parallel tasks are defined in lines 5-9. Each data type has one or more data distribution types. SEParAT supports a predefined set of regular data types, e.g., arbitrary multi-dimensional arrays, and regular data distribution types, such as block-cyclic distributions. The parallel tasks are defined by specifying a set of input and output parameters with the corresponding data types and a set of implementation variants. Each implementation variant defines the data distribution types of the parameters of the corresponding parallel task. Implementations of basic parallel tasks additionally contain

```
1  <!-- definition of external parameters -->
2  <ProblemParam Name="n" DefaultValue="1024"/>
3  <MachineParam Name="t_C"/> <!-- computing power -->
4
5  <!-- data type and data distribution type definitions -->
6  <DataType Name="myMatrix" DataType="matrix"
7            C-Type="double" Dimension="2" Size="n;n">
8    <DataDistrib Name="block" Description="BLOCK"/>
9  </DataType>
10
11 <!-- example for a basic parallel task definition -->
12 <Module Name="myNode" Id="1">
13   <Param Name="in"  Id="1" Type="myMatrix"/>
14   <Param Name="out" Id="2" Type="myMatrix"/>
15   <Implementation Name="module1_block" Id="1">
16     <Distrib ParamRef="1" Type="block"/>
17     <Distrib ParamRef="2" Type="block"/>
18     <BasicModule>
19       <Runtime Formula="T_par(p,n,t_C)=0.1*t_C*n^2
20                         +(0.9*t_C*n^2)/p"/>
21     </BasicModule>
22   </Implementation>
23 </Module>
24
25 <!-- example for a complex parallel task definition -->
26 <Module Name="task graph" Id="2">
27   <Param Name="in"  Id="1" Type="myMatrix"/>
28   <Param Name="out" Id="2" Type="myMatrix"/>
29   <Implementation Name="main impl" Id="1">
30     <Distrib ParamRef="1" Type="block"/>
31     <Distrib ParamRef="2" Type="block"/>
32     <ComplexModule>
33       <!-- nodes of the task graph -->
34       <StartNode Name="entry" Id="1"/>
35       <Node Name="myNode#1" Id="2" ModuleRef="1"/>
36       <StopNode Name="exit" Id="3"/>
37       <!-- edges of the task graph -->
38       <Edge Id="1" SourceNodeId="1" SourceParamId="1"
39                    TargetNodeId="2" TargetParamId="1"/>
40       <Edge Id="2" SourceNodeId="2" SourceParamId="2"
41                    TargetNodeId="3" TargetParamId="2"/>
42     </ComplexModule>
43   </Implementation>
44 </Module>
45
46 <!-- definition of the root complex parallel task -->
47 <MainModule ModuleRef="2"/>
```

Fig. 5 Example for an input file describing a parallel application task graph.

a symbolic runtime formula that defines the execution time depending on the number of processors p, see lines 18-21. Lines 32-42 show the implementation of a complex parallel task that contains an application task graph consisting of a set of nodes and a set of edges. Each node refers to a parallel task defined in the input and each edge connects an output parameter of the source node with an input parameter of the target node.

The properties of a homogeneous platform are provided in form of constants, e.g., the time needed to execute an arithmetic operation, and functions, e.g., the time required to

```
<!-- specification of the homogeneous subclusters -->
<Machines>
<Machine Id="0" Name="Subcluster1" Processors="8">
  <Constant Name="t_C" Value="6.9E-8"/>
  <Constant Name="t_S" Value="2.0E-6"/>
  <Constant Name="t_B" Value="1.2E-9"/>
  <Function Name="t_sendrecv"
    Formula="t_sendrecv(n,t_S,t_B) = t_S + t_B * n"/>
</Machine>
<Machine Id="1" Name="Subcluster2" Processors="16">
  <!-- properties of Subcluster2 -->
</Machine>
</Machines>

<!-- specification of the interconnections between
     the subclusters -->
<MachineConnections>
<Connection Endpoint1="0" Endpoint2="1">
  <Constant Name="t_S1" Value="0.000002"/>
  <Constant Name="t_S2" Value="0.000004"/>
  <Constant Name="t_B"  Value="0.0000000012"/>
</Connection>
</MachineConnections>
```

Fig. 6 Example for the specification of a parallel platform.

```
<ProblemParamDesc>
  <!-- problem size -->
  <Constant Name="n" Value="1024"/>
</ProblemParamDesc>
```

Fig. 7 Example for an input file providing additional information.

execute a broadcast operation depending on the number of participating processors and the amount of data to be transmitted. A heterogeneous platform specification consists of a set of definitions of homogeneous platforms and the definition of the communication performance of the interconnection between each pair of homogeneous subclusters. An example for the specification of a heterogeneous platform consisting of two homogeneous subclusters is shown in Fig. 6.

The additional information is provided in a third input file. This file consists of a collection of user-defined constants and functions that may be used in the data types and symbolic runtime formulas specified with the application task graph. An example for such an input file is given in Fig. 7. SEParAT combines the information given in the task graph specification and in the additional information input file into an annotated task graph that is used in further processing steps.

4.1.2 Output structure of SEParAT

The output of SEParAT is given in a file output as well as a visual output. The file output contains a feasible schedule for a specific application on a specific target platform. The schedule produced can be processed by other tools, e.g., to execute the parallel tasks on the specified platform, or to generate coordination code that controls the execution of the parallel application. Prior to producing the output file an optional validation step can be

```
1  <Schedule Id="2" Makespan="0.0287">
2    <DataRedistribution Id="1" Name="E(1)[1->2]"
3                        StartTime="0.0" FinishTime="0.00519">
4      <SourceProcessorGroup>1 2 3 4 5 6 7 8
5      </SourceProcessorGroup>
6      <TargetProcessorGroup>1 2 3 4</TargetProcessorGroup>
7    </DataRedistribution>
8    <ModuleCall Id="2" Name="myNode#1" ModuleRef="1"
9                ImplementationRef="1"
10               StartTime="0.00519" FinishTime="0.0287">
11     <ProcessorGroup>1 2 3 4</ProcessorGroup>
12   </ModuleCall>
13 </Schedule>
```

Fig. 8 Example for a schedule produced by SEParAT.

performed. For each node and data dependency edge, the output file contains the following information: the start time of the node or data dependency edge, the processor group(s) on which the node or data dependency edge is executed, the finish time of the node or data dependency edge, and the module variant of each node that should be used when executing the parallel application.

An example is shown in Fig. 8. Lines 2-7 contain a data re-distribution operation that transfers data from processor group {1,...,8} to processor group {1,2,3,4} in the time interval from `0.0` to `0.00519`. Lines 8-12 define the execution of the node `myNode#1` on processor group {1,2,3,4} from time `0.00519` to time `0.028704`. It has a type of myNode (`ModuleRef=1`) and its module variant module1_block (`ImplementationRef=1`) is used. The `ModuleRef` and `ImplementationRef` refer to the `Id` specified in the module definition and the implementation definition, respectively, in the input file, see Fig. 5.

When running SEParAT as a stand-alone application, visual output is also possible and includes a computed schedule as Gantt-chart, intermediate results, e.g., the layered task graph of layer-based scheduling algorithms [14], or comparison charts. This is important to track the scheduling process and to identify performance bottlenecks.

4.2 Internal processing and transformation steps

SEParAT provides internal mechanisms to compute a feasible schedule for a given scheduling problem, and to validate computed schedules. Also, large sets of scheduling problems for the comparison and analysis of scheduling algorithms can be created.

4.2.1 Formula evaluation

The formula evaluator handles the symbolic runtime formulas used to define computation and communication costs. This component stores all formulas, variables, and constants of the scheduling problem and provides support for the verification, simplification, and evaluation of formulas.

4.2.2 Graph generation

SEParAT provides support for the creation of synthetic scheduling problems with user-defined parameters. These parameters can influence the structure of the task graph, the

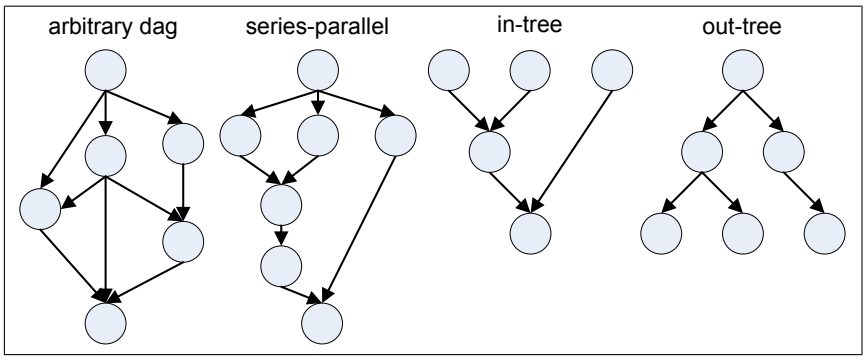

Fig. 9 Graph structures supported by SEParAT. If the graph structure in-tree (out-tree) is chosen, an extra entry node (exit node) is added to generate a valid task graph.

amount of module variants created, and the cost model used. As graph structures series-parallel graphs, in-trees, out-trees, and arbitrary directed acyclic graphs (dags) are supported, see Fig. 9. Series-parallel graphs are a common representation of parallel applications consisting of sequential and parallel parts. The tree structures represent divide-and-conquer algorithms where the control flow proceeds bottom-up (in-tree) or top-down (out-tree). Each graph structure can be parameterized with the number of dependencies to create and the information whether to prefer flat or deep task graphs.

The cost models supported by SEParAT include unit costs (identical costs for all parallel tasks), table costs (the costs are read from a predefined table) and random costs (the costs are completely random). Additionally, there are two more sophisticated models:

– In Amdahl's model the computation costs are defined using the formula

$$T_{par}(P) = \alpha * T_{sequ} + (1-\alpha) * \frac{T_{sequ}}{P}$$

where $T_{par}(P)$ is the parallel execution time of a parallel task on P processors, α is the sequential fraction of the parallel task, and T_{sequ} is the sequential execution time of the parallel task. The sequential fraction and the sequential execution time are selected randomly inside a user-defined range.

– In the model used in [28, 19], three different types of computation with different computational complexities are considered; image processing ($b*n$), array sorting ($b*n*\log(n)$) and matrix-matrix multiplication ($b*n^{\frac{3}{2}}$). The parameter b is picked randomly from a user-defined range. The value n is determined by the data sizes of the input parameters. The sequential execution time is computed by multiplying the resulting values with the inverse of the flop rate of the parallel target platform. The parallel execution time is computed via Amdahl's model and a random sequential fraction for each parallel task.

4.2.3 Graph transformation

Internally, SEParAT transforms an annotated task graph that was either built up from input files or generated randomly into a simple task graph, which is then the input of the scheduling algorithm library. The simple task graph is flat, i.e., it contains only basic parallel tasks

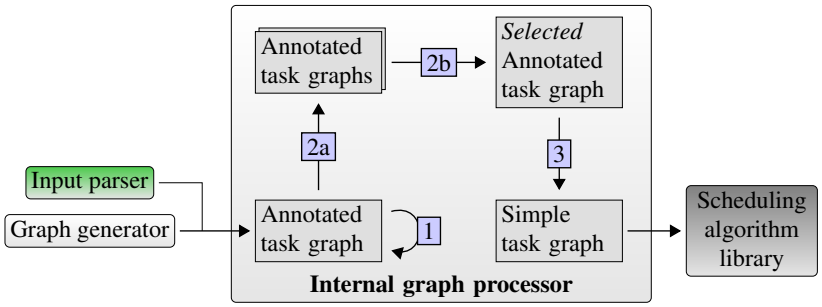

Fig. 10 Transformation of an annotated task graph into a simple task in four steps: (1) removal of hierarchies, (2a and 2b) selection of a specific module variant for each parallel task, and (3) creation of the simple task graph.

with dependencies, and defines a single module variant for each parallel task. The processing steps performed are illustrated in Fig. 10.

First, the hierarchies of an annotated task graph are removed by a recursive algorithm that runs over the nodes of the annotated task graph. When the algorithm encounters a complex node, it first handles the node recursively, and afterwards replaces the node with the corresponding task graph (excluding the entry and the exit nodes). The edges are adjusted accordingly.

Step (2a) converts the resulting flat task graph into a set of annotated task graphs such that each node has only a single module variant. This step assumes that the same module variant is used for all parallel tasks with the same type. Based on this assumption, all possible combinations of module variants for parallel tasks with different types are considered. Since an application usually contains only a small number of different parallel tasks, the number of task graphs constructed is reasonable.

Afterwards, one of the annotated task graphs is selected based on a user-defined criterion (2b). Examples for such a criterion are to use the smallest execution time of each node's module variant on one processor, or to determine the best matching for the data distributions of data dependent nodes. At last, step (3) creates the simple task graph by computing the data re-distribution costs for the data dependency edges based on the module variants chosen.

4.2.4 Scheduling algorithm library

The scheduling algorithm library of SEParAT provides a set of 16 scheduling algorithms for homogeneous parallel machines as well as heterogeneous parallel machines. Additional user-defined scheduling algorithms can be added to the library.

Four categories of scheduling algorithms are provided for homogeneous parallel target platforms: basic algorithms, allocation-and-scheduling-based algorithms, layer-based algorithms, and configuration-based algorithms, see also the overview given in Sect. 2. The two ***basic algorithms*** Data and Task compute a pure data parallel schedule, or a schedule where each parallel task is restricted to run on a single processor, respectively. These algorithms can be used to show the benefit of a mixed parallel execution. ***Allocation-and-scheduling-based algorithms*** are CPA [30], CPR [29], TSAS [31], and the approximation algorithms [22] and [21]. The allocation phase and the scheduling phase are implemented separately, see Fig. 11 (top) for an illustration. ***Layer-based algorithms*** are TwoL-Level [32], TwoL-Tree [33], Approx-2 [24], Approx-$\sqrt{3}$ [25], and Approx-$\frac{3}{2}$ [26]. These al-

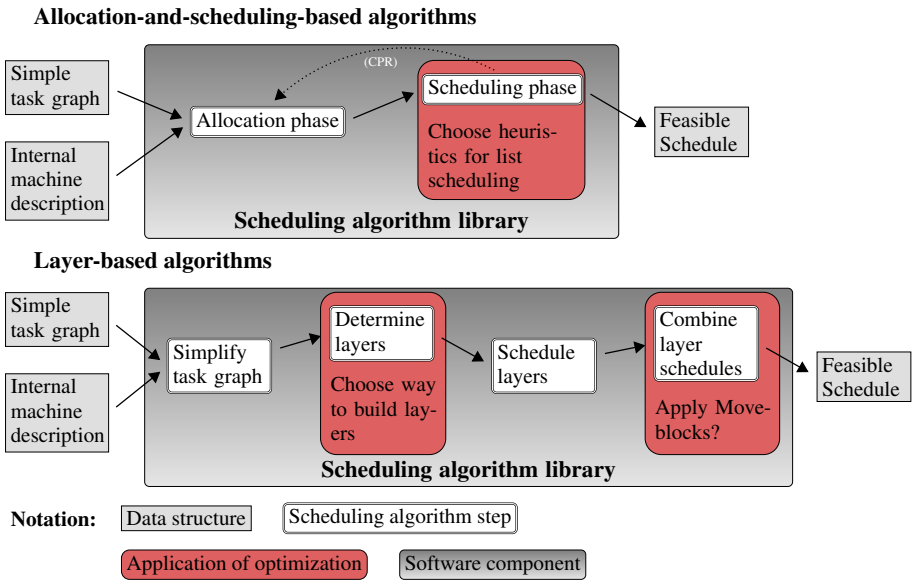

Fig. 11 Phases executed by allocation-and-scheduling-based algorithms (top) and layer-based algorithms (bottom). The phases where an optimization can be applied are highlighted in red.

gorithms proceed in four steps, see Fig. 11 (bottom). ***Configuration-based algorithms*** use predefined configurations of processors to schedule the task graph in a single step [7]. To achieve good results the definition of the configurations is important. In SEParAT, the configurations can be specified based on fractions of the number of available processors. The fifth category of scheduling algorithms consists of algorithms designed for heterogeneous target platforms. The algorithms H-CPA [27], M-HEFT1 [9], and M-HEFT2 [9] belong to this category.

SEParAT supports various optimizations and fine tunings of the scheduling algorithms implemented, see Fig. 11. The scheduling phase of the allocation-and-scheduling-based algorithms is based on a modified list scheduling algorithm that uses a priority function defining the order in which the parallel tasks are scheduled. SEParAT supports different priority functions for this step, e.g., smallest or largest execution time first, earliest possible start time first, smallest bottom/top level first, and largest number of successors first. For layer-based scheduling algorithms there are two optimizations implemented. First, the decomposition of the task graph into layers of independent parallel tasks can be influenced by the restriction of the number of nodes per layer and the selection of a decomposition heuristic. The second optimization is the application of the algorithm Move-blocks [19], which is able to reduce the makespan of layer-based scheduling algorithms by a smart combination of the schedules computed for each layer into the final schedule for the entire task graph.

4.2.5 Validation

SEParAT supports the validation of computed schedules and, thus, helps to ensure the correct execution of user-implemented scheduling algorithms. The validation process also gathers information that help developers to get further insights into the schedules computed.

These information include the overall number of data re-distributions, the total costs for the data re-distribution operations, the number of consecutive parallel tasks that are executed on the same processor group, and the communication to computation ratio.

4.3 User interface

The user interface allows a direct interaction with SEParAT (graphical user interface) or the automated execution of the scheduling process (command line interface). The graphical user interface controls the input and output, and displays the annotated task graph, the internal machine description as well as additional information. Intermediate results, computed schedules, validation results of computed schedules, and comparison charts can also be visualized.

Figure 12 shows some aspects that are covered by the graphical user interface. An annotated task graph as it is created from the input specification is shown in Fig. 12 (a). A layered task graph, which is an intermediate result of a layer-based scheduling algorithm, is shown in Fig. 12 (b). Inner nodes of the task graph with the same color belong to one layer. Figure 12 (c) shows a computed schedule with unit scale time axis. The schedule contains data re-distribution operations (green) and activations of parallel tasks (yellow tones for the first layer, and gray tones for the second layer). The unit time scale allows the visualization of schedules with highly varying execution times by considering the start time and finish time of node and edge executions as discrete events. An example for a comparison chart is given in Fig. 12 (d) using five scheduling algorithms (from left to right: Task, Data, CPA, TwoL-Level, and Approx-2). The scheduling algorithms have been applied to a generated task graph with 50 nodes and a homogeneous parallel machine with 16 processors. SEParAT also supports the comparison of the optimizations applied to the scheduling algorithms by a suitable visualization.

The command line interface of SEParAT is intended for an automated execution, e.g., for the cooperation with external tools or for running benchmarks. In this case, SEParAT is controlled via two files. The first file includes configuration options for the scheduling algorithm to be used, e.g., the optimizations that should be used. An example is shown in Fig. 13. The second file specifies the input and output, e.g., the set of input files for a benchmark run. A simplified example is shown in Fig. 14. The information provided lead to benchmarks for task graphs with 100 to 1000 nodes with a step size of 100. For each number of nodes 400 different task graphs are considered, which are read from the directory `workspace/testSet_ALL_extended_format`.
The considered target platforms comprise 8 to 32 processors with a step size of 8. The results are stored to the file `test.out`.

5 Experimental evaluation

In this section, different aspects of SEParAT are evaluated. The aspects covered in Subsect. 5.1 are the time needed to read in different types of scheduling problems, the memory consumption of the internal graph data structures, and the execution time of a scheduling pass for two scheduling algorithms. Results concerning the scheduling algorithms are given in Subsect. 5.2.

Benchmark results were obtained by compiling SEParAT with Java 1.6.0_20 (64 bit version) and running it on an AMD Opteron Dual-Core “Egypt” system clocked at 1.8 GHz

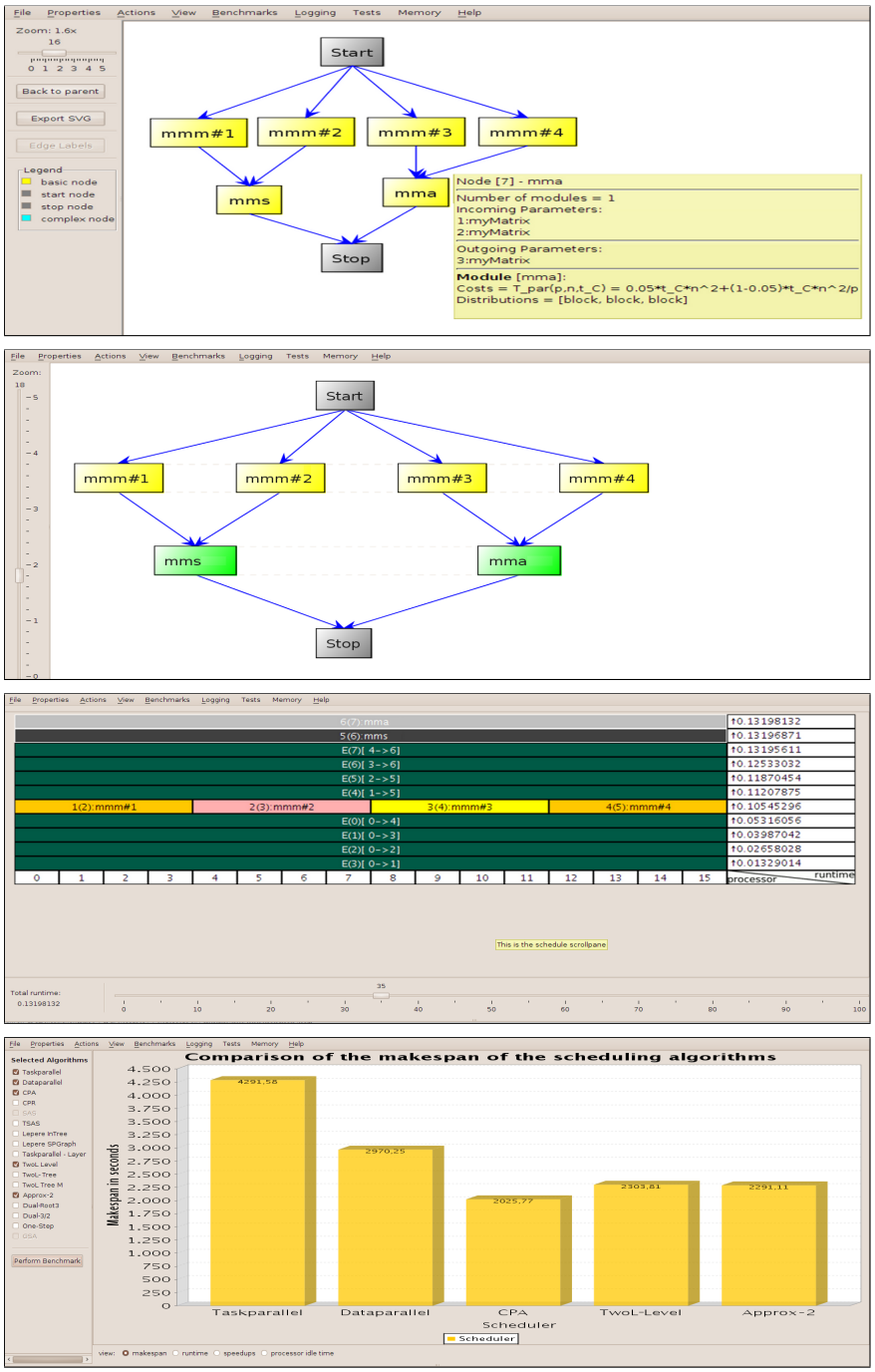

Fig. 12 Examples of the display in the graphical user interface of SEParAT: (a) an annotated task graph, (b) a layered task graph, which is an intermediate result of layer-based scheduling algorithms, (c) a schedule with unit scale time axis, and (d) a makespan comparison chart.

```
# select a layer-based algorithm (category 1 out of 5)
algorithmCategory=1
# select algorithm TwoL-Level (algorithm 10 out of 16)
algorithm=10
# enable the computation of data re-distribution costs
considerRedistributionCosts=true
# construct layers based on a topological sort
# (layerBuilder 1 out of 4)
layerBuilder=1
# Combine layer schedules with the Move-blocks algorithm
moveBlocks=true
```

Fig. 13 Example for the specification of scheduling algorithm options.

```
graphsPerNodeNumber=400
nodesFrom=100
nodesStep=100
nodesTo=1000
outputFileName=test.out
procsFrom=8
procsStep=8
procsTo=32
testSetDir=workspace/testSet_ALL_extended_format
```

Fig. 14 Example for the specification of benchmarking options.

and 4 GiB RAM. The task graphs were synthetic task graphs generated by SEParAT. The graph type was chosen as series-parallel graph and the runtime model was based on the model in [28]. The number of nodes in a task graph ranged from 50 to 1000. For each number of nodes, 400 different task graphs were created and the arithmetical mean was computed and reported in the following figures.

5.1 Evaluation of SEParAT

The following results show that the overhead in terms of time and memory is small when using the framework SEParAT. Figure 15 shows the time needed to read in generated task graphs with different numbers of module variants for the nodes. The maximum number of module variants per node was set to three. The fraction of module variants was set to 0.1, 0.5, and 0.9 resulting in task graphs where each 10th, 2nd and almost all nodes had more than one module variant. The time needed to read in the task graphs is linear in the number of nodes. The results show that reading in task graphs is fast. For task graphs with 1000 nodes and a high fraction of module variants it took less than one second.

Figure 16 shows the memory consumption for the annotated task graphs (top) and simple task graphs for the homogeneous case (bottom left) as well as the heterogeneous case (bottom right). The task graphs considered were generated with a fraction of module variants of 0.5 and at most three module variants per node were created. The size of the input files of the parallel application task graphs ranged from 0.1 MByte in average for task graphs with 100 nodes to 1.13 MByte in average for task graphs with 1000 nodes.

The memory consumption for the annotated task graphs was small with at most 3.24 MByte in average for graphs with 1000 nodes. The transformation of an annotated task graph into a simple task graph selects a single module variant for each parallel task and

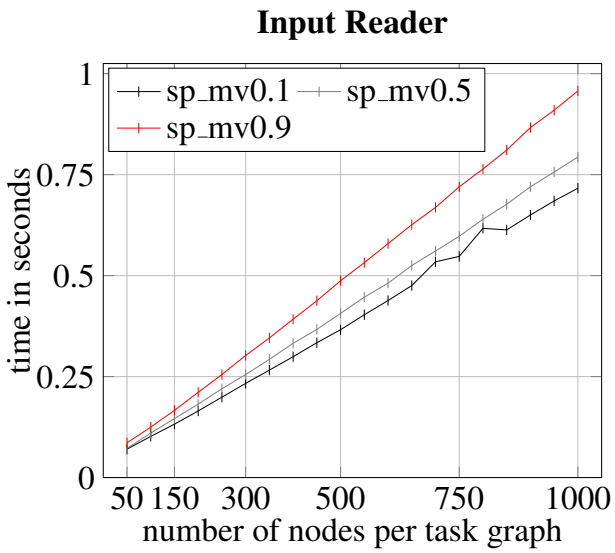

Fig. 15 Time needed to read in generated task graphs of type series-parallel graph. The task graphs comprise nodes with a low (sp_mv0.1), a medium (sp_mv0.5), and a high (sp_mv0.9) fraction of module variants.

removes information on the input and output parameters of the parallel tasks. The memory usage of a simple task graph depends on the type of target platforms. For homogeneous target platforms, the cost information is stored in a lookup table of size $P \times N$ where P is the number of processors of the platform and N is the number of nodes in the task graph. As a consequence, the memory consumption depends also on the number of processors, see Fig. 16 (bottom left). SEParAT does not construct such a cost table for heterogeneous platforms and, thus, the amount of memory used is smaller than for homogeneous platforms and independent from the number of processors, see Fig. 16 (bottom right). The maximum memory consumption measured was 4.07 MByte for a simple task graph with 1000 nodes on a homogeneous platform with 512 processors.

Figure 17 shows the time needed to execute an entire scheduling pass for the scheduling algorithms Task and CPA. A scheduling pass comprises the parsing of the input file, the internal transformations, the scheduling, and the generation of an output file. The task graphs consisted of 100 to 1000 nodes with a step size of 100. Homogeneous target platforms with a low number of processors (8 and 32), a medium number of processors (128), and a high number of processors (512) were used. The scheduling algorithm Task is based on a very simple mechanism and, thus, mainly reflects the overhead resulting from the handling of the input and output files, and the creation of the internal data structures. For a low number of processors the execution time of the scheduling pass was small and resulted in execution times between 150 milliseconds (100 nodes, 8 processors) and 1.55 seconds (1000 nodes, 32 processors). In average, the time to execute the scheduling pass took approximately 750 milliseconds for both numbers of processors. For larger parallel target platforms the execution time was 818 milliseconds (medium number of processors) and 1052 milliseconds (high number of processors) in average.

The scheduling algorithm CPA uses a more sophisticated mechanism to compute the schedules. However, for small numbers of processors the execution time of the scheduling

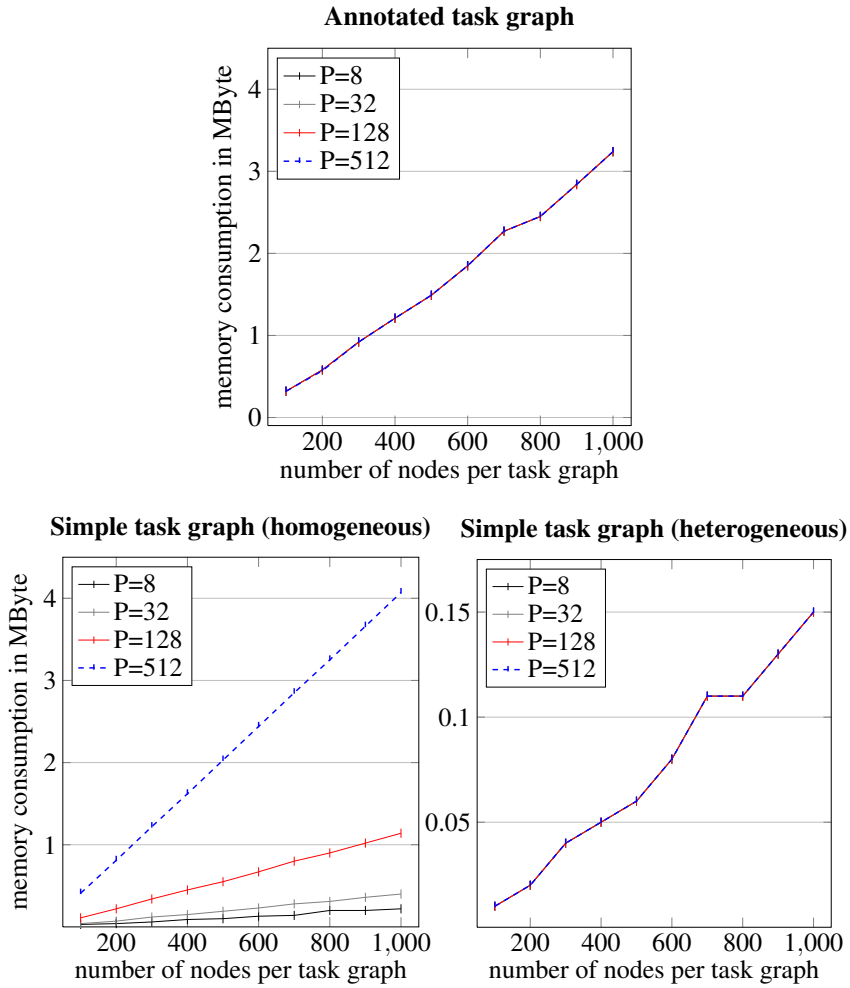

Fig. 16 Memory consumption of the graph data structures depending on the number of nodes per task graph and on the number of processors of a parallel target platform.

pass is similar to that of the scheduling algorithm Task. This result shows that for small numbers of processors the overhead for additional operations beside the scheduling prevails. For a higher number of processors the time needed for executing the scheduling algorithm CPA became much higher. On a medium number of processors the execution of the scheduling pass took 921 milliseconds in average and for a high number of processors it took 2075 milliseconds. This shows that the impact of the overhead of the additional operations becomes less relevant when considering large parallel target platforms. Nevertheless, the time needed to execute the scheduling pass is reasonable for static scheduling, since usually the schedule is computed only once and used for several executions of the mixed parallel application afterwards.

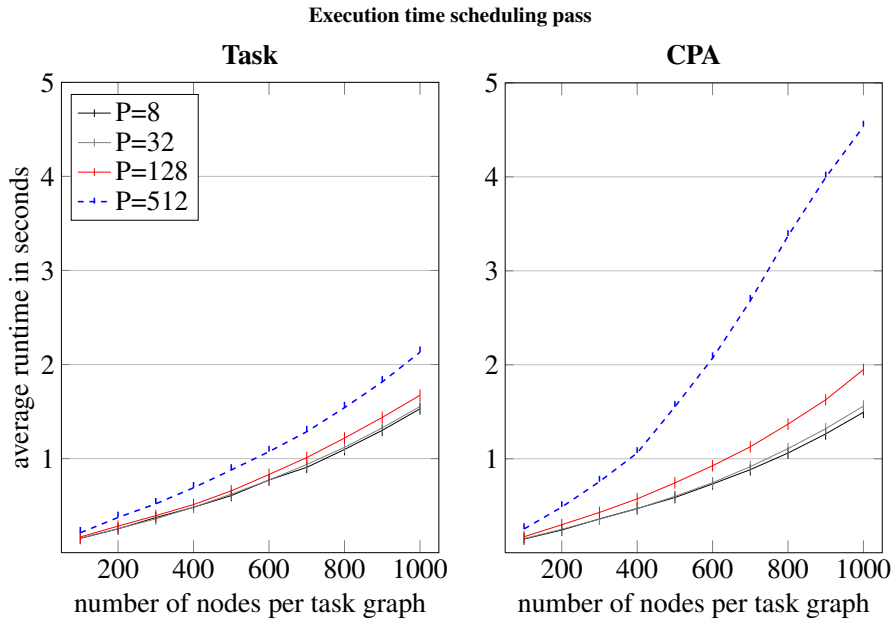

Fig. 17 Execution time of a scheduling pass for the scheduling algorithms Task and CPA. Task graphs with 100 to 1000 nodes were considered and schedules for homogeneous target platforms with 8, 32, 128, and 512 processors were computed.

5.2 Comparison of scheduling algorithms

In the following, a comparison of selected scheduling algorithms for homogeneous parallel machines in terms of executions time and makespans of the schedules computed is presented. The comparison uses the algorithms Task, Data, CPA, TSAS, TwoL-Level, Approx-2, and OneStep without any optimizations. For the benchmarks, 400 different series-parallel task graphs have been created for each number of nodes in the range from 50 to 350 with a step size of 50. The arithmetical mean of the results for each number of nodes is reported.

Figure 18 shows the runtimes of the scheduling algorithms for homogeneous target platforms with 8, 128, and 512 processors. For all scheduling algorithms, the runtime increases with an increasing number of nodes per task graph and fixed number of processors. Except for TSAS and OneStep, the runtime also increases with an increasing number of processors and a fixed number of nodes. The runtime of TSAS mainly depends on the convergence rate of the convex optimization problem that has to be solved in the allocation step.

The results show that the simple algorithms Data and Task have the lowest runtimes. The scheduling algorithm Task needed between 9.5 milliseconds (8 processors) and 15.3 milliseconds (512 processors) in average. The scheduling algorithm Data needed between 9.4 milliseconds (8 processors) and 32.2 milliseconds (512 processors) in average. The layer-based algorithms TwoL-Level and Approx-2 needed approximately the same time to compute a schedule. The reason is the overhead of the phases (i), (ii), and (iv) that are identical for all layer-based algorithms. Both scheduling algorithms were just slightly slower compared to the simple algorithms. With average runtimes between 22.4 milliseconds and 91.9 milliseconds (TwoL-Level) and between 22.8 milliseconds and 103.7 milliseconds

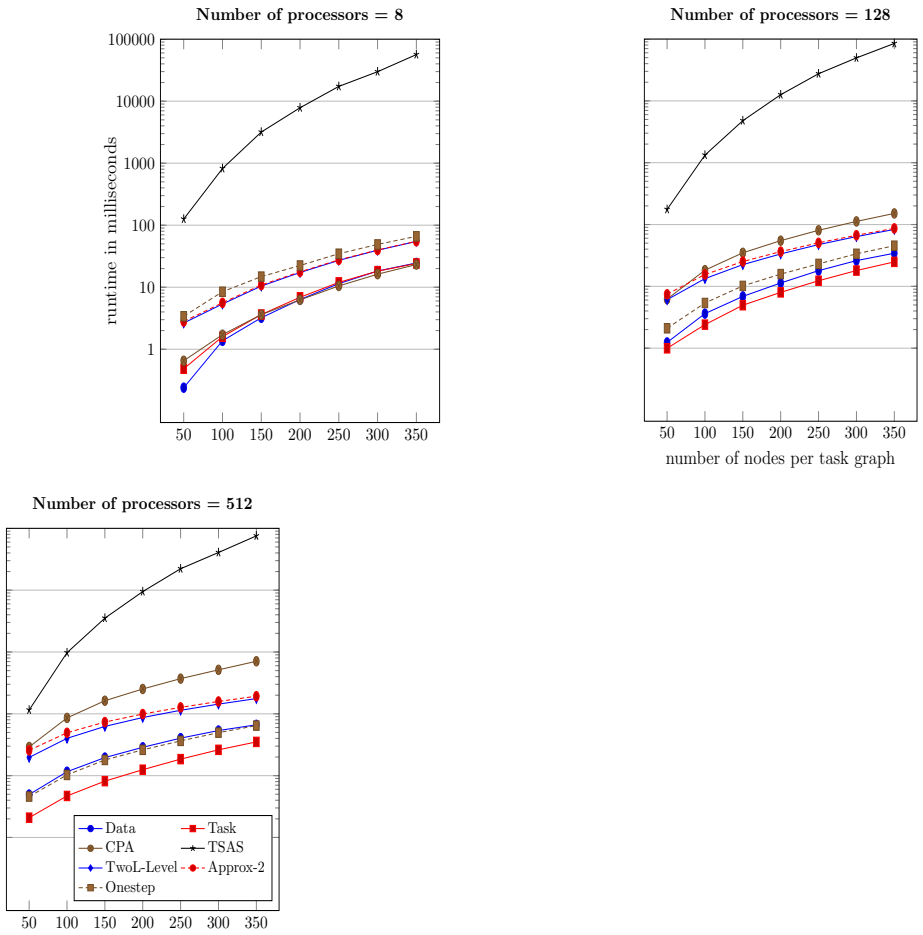

Fig. 18 Runtimes of the scheduling algorithms for task graphs with 50 up to 350 nodes and homogeneous target platforms with 8, 128, and 512 processors.

(Approx-2) they were still fast. The allocation-and-scheduling-based algorithms CPA and TSAS require the highest runtime. Although CPA is competitive for target platforms with 8 processors its performance decreases considerably with an increasing number of processors. The scheduling algorithm TSAS was extremely slow. The average runtime ranges from 16.5 seconds (8 processors) to 25.8 seconds (128 processors). The runtime of the configuration-based algorithm OneStep was the second highest when the number of processors of the target platform was 8. For target platforms with 128 and 512 processors, the runtime of OneStep was similar to that of the simple algorithms.

Figure 19 shows the makespan of the computed schedules for homogeneous target platforms with 8, 128, and 512 processors. The results show that the makespan increases when the number of nodes per task graph increases and drops when the number of processors is increased. An exception is CPA where the makespan increases when the number of processors of the target platform is increased from 128 to 512. This is due to the relatively simple

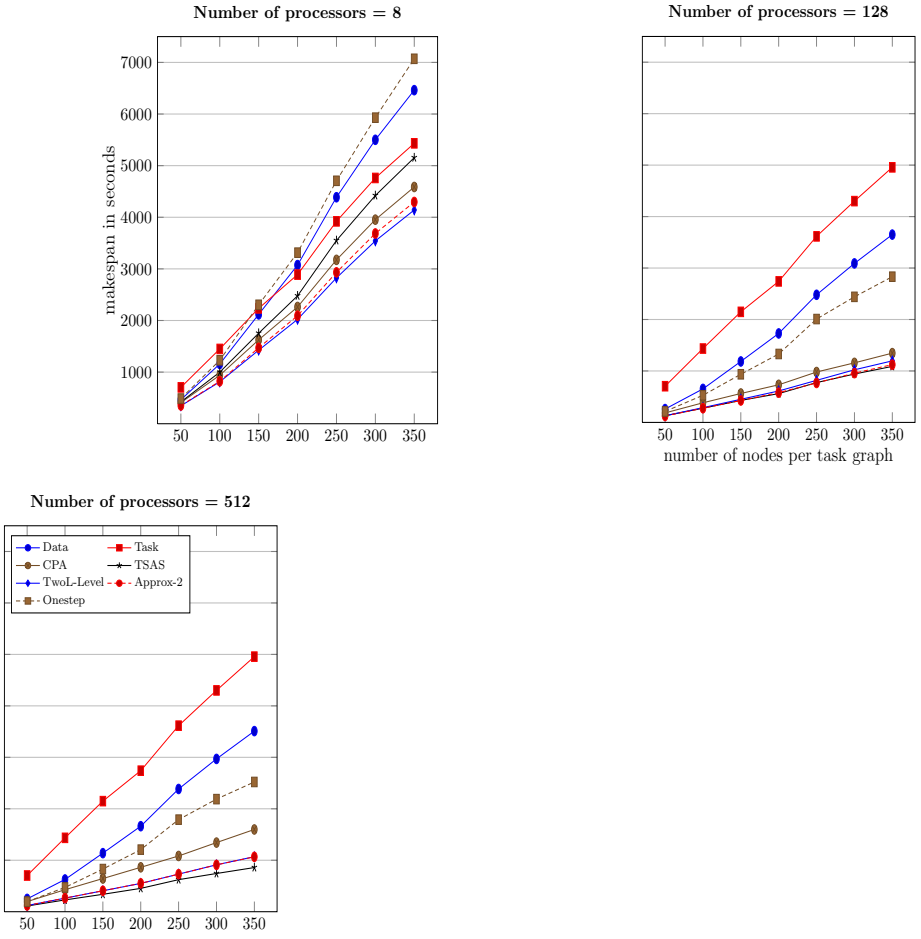

Fig. 19 Makespan of the computed schedules for task graphs with 50 up to 350 nodes and target platforms with 8, 128, and 512 processors.

approach used to determine the number of processors for each node, see also [5] which proposes an improvement for this behavior.

The disadvantage of the algorithms Data and Task is the only slight improvement when the target platform was changed from 128 to 512 processors. The makespans produced by OneStep are also much higher than that of the allocation-and-scheduling algorithms and the layer-based algorithms. This may be improved with a scheduling problem specific way to build the configurations.

The layer-based algorithms produce very good makespan results. For a target platform with 8 processors, TwoL-Level creates the lowest makespan followed by Approx-2. For target platforms with 128 and 512 processors, only TSAS computes schedules with a lower makespan. The average makespan of the schedules computed by TSAS for a target platform with 512 processors is 479 seconds. In contrast, the average makespan of the simple algo-

rithms Data and Task is 1791 seconds and 2843 seconds, respectively. This illustrates the improvement capabilities when choosing an appropriate scheduling algorithm.

The computation of a feasible schedule with minimal makespan depends highly on the scheduling problem and the scheduling algorithm chosen. As can be seen from the runtime and makespan results the algorithm that computes the best result varies. Although, allocation-and-scheduling-based algorithms compute very good schedules they need much more time to execute the scheduling process. The layer-based algorithms have shown to compute competitive schedules with a comparatively low runtime for the synthetic task graphs considered.

6 Conclusion

The computation of schedules that allow an efficient execution of parallel application task graphs is a crucial point when dealing with large mixed parallel applications. The support by a scheduling environment enables developers to avoid the error-prone and complex determination of such a schedule by hand. In this article, we have proposed SEParAT, which is a flexible scheduling framework offering a large variety of features needed when solving scheduling problems. The fully automated execution allows the usage of SEParAT as additional component for other parallel programming tools that include a scheduling step. The user driven execution allows the analysis of scheduling algorithm behavior for given scheduling problems. The design of the component-based software-architecture provides the benefit to extend/substitute components and to use the components within other tools. The key component of SEParAT is the scheduling algorithm library for homogeneous as well as heterogeneous target platforms. The application of different optimizations to the scheduling algorithms is a new feature.

A detailed evaluation of SEParAT has shown that SEParAT allows fast scheduling without adding a significant overhead and that it is applicable to large parallel application task graphs. A selection of the scheduling algorithms has been compared in terms of the runtime of the scheduling algorithms and the parallel execution time of the computed schedules. The results show that the mixed parallel approach outperforms a pure data and a pure task parallel execution. However, the results also show that the selection of an advantageous scheduling algorithm for a parallel application task graph depends on both the specific parallel application and target platform and, thus, is a tedious process. As a consequence, support by SEParAT is especially valuable in these cases.

Acknowledgements This project was supported by Deutsche Forschungsgemeinschaft (DFG) grants RU591/9-1 and RU591/9-2.

References

1. I. Ahmad, Y.-K. Kwok, M.-Y. Wu, and W. Shu. Casch: A tool for computer-aided scheduling. *IEEE Concurrency*, 8:21–33, 2000.
2. A. Alexandrov, M.F. Ionescu, K.E. Schauser, and C. Scheiman. LogGP: Incorporating long messages into the LogP model for parallel computation. *Journ. of Parallel and Distr. Computing*, 44(1):71–79, 1997.
3. L. Baduel, F. Baude, D. Caromel, A. Contes, F. Huet, M. Morel, and R. Quilici. *Grid Computing: Software Environments and Tools*, chapter Programming, Deploying, Composing, for the Grid, pages 205 – 229. Springer-Verlag, January 2006.

4. 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 Comp.*, 28(10):37–47, 1995.
5. 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.
6. K.J. Barker, K. Davis, A. Hoisie, D.J. Kerbyson, M. Lang, S. Pakin, and J.C. Sancho. Using Performance Modeling to Design Large-Scale Systems. *IEEE Computer*, 42(11):42–49, 2009.
7. V. Boudet, F. Desprez, and F. Suter. One-step algorithm for mixed data and task parallel scheduling without data replication. In *IPDPS '03: Proc. of 17th Int. Symp. on Parallel and Distr. Processing*, page 41.2. IEEE Comp. Society, 2003.
8. J. Cao, A. T. Chan, Y. Sun, S. K. Das, and M. Guo. A taxonomy of application scheduling tools for high performance cluster computing. *Cluster Computing*, 9(3):355–371, 2006.
9. H. Casanova, F. Desprez, and F. Suter. From Heterogeneous Task Scheduling to Heterogeneous Mixed Parallel Scheduling. In Marco Danelutto, Domenico Laforenza, and Marco Vanneschi, editors, *Proc. of 10th Int. Euro-Par Conf. (Euro-Par'04)*, volume 3149 of *LNCS*, pages 230–237, Pisa, Italy, August/September 2004. Springer.
10. D.E. Culler, R.M. Karp, D.A. Patterson, A. Sahay, K.E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a realistic model of parallel computation. In *Principles Practice of Parallel Programming*, pages 1–12, 1993.
11. J. Du and J.Y.-T. Leung. Complexity of Scheduling Parallel Task Systems. *SIAM Journal on Discrete Mathematics*, 2(4):473–487, 1989.
12. J. Dümmler, R. Kunis, and G. Rünger. A Comparison of Scheduling Algorithms for Multiprocessor-tasks with Precedence Constraints. In *Proc. of the 2007 High Performance Computing & Simulation (HPCS'07) Conference*, pages 663–669. ECMS, 2007.
13. J. Dümmler, R. Kunis, and G. Rünger. A Scheduling Toolkit for Multiprocessor-Task Programming with Dependencies. In *Proc. of the 13th International Euro-Par Conference*, volume 4641 of *LNCS*, pages 23–32. Springer, 2007.
14. J. Dümmler, R. Kunis, and G. Rünger. Layer-Based Scheduling Algorithms for Multiprocessor-Tasks with Precedence Constraints. In *Parallel Computing: Architectures, Algorithms and Applications: Proc. of the Int. Conf. ParCo 2007*, volume 15 of *Advances in Parallel Computing*, pages 321–328. IOS Press, 2007.
15. J. Dümmler, T. Rauber, and G. Rünger. A Transformation Framework for Communicating Multiprocessor-Tasks. In *Proc. of the 16th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP '08)*, pages 64–71. IEEE, 2008.
16. E. Frachtenberg and U. Schiegelshohn. New Challenges of Parallel Job Scheduling. In Eitan Frachtenberg and Uwe Schwiegelshohn, editors, *Proceedings of the 13th Job Scheduling Strategies for Parallel Processing*, volume 4942 of *Lecture Notes in Computer Science (LNCS)*, pages 1–23. Springer, April 2008.
17. M. Hill, W. McColl, and D. Skillicorn. Questions and Answers about BSP. *Scientific Programming*, 6(3):249–274, 1997.
18. M. Kühnemann, T. Rauber, and G. Rünger. Performance modelling for task-parallel programs. *Performance Analysis and Grid Computing*, pages 77–91, 2004.
19. R. Kunis and G. Rünger. Optimizing layer-based scheduling algorithms for parallel tasks with dependencies. *Concurrency and Computation: Practice and Experience*, 23(8):827–849, 2011.
20. O.-K. Kwon, J. Hahm, S. Kim, and J. R. Lee. Grasp: A grid resource allocation system based on ogsa. In *13th Int. Symp. on High-Performance Distr. Comp.*, pages 278 – 279, 2004.
21. R. Lepere, G. Mounie, and D. Trystram. An approximation algorithm for scheduling trees of malleable tasks. *European Journ. of Operational Research*, 142:242–249, 2002.
22. R. Lepere, D. Trystram, and G.J. Woeginger. Approximation algorithms for scheduling malleable tasks under precedence constraints. *Int. Journ. of Foundation of Comp. Sci.*, 13(4):613–627, 2002.
23. T. Lewis and H. El-Rewini. Parallax: A tool for parallel program scheduling. *IEEE Parallel Distr. Technol.*, 1(2):62–72, 1993.
24. W. Ludwig and P. Tiwari. Scheduling Malleable and Nonmalleable Parallel Tasks. In *SODA '94: Proc. of 5th annual ACM-SIAM Symp. on Discrete Algorithms*, pages 167–176. SIAM, 1994.
25. G. Mounie, C. Rapine, and D. Trystram. Efficient approximation algorithms for scheduling malleable tasks. In *SPAA '99: Proc. of 11th annual ACM Symp. on Parallel algorithms and architectures*, pages 23–32. ACM Press, 1999.
26. G. Mounie, C. Rapine, and D. Trystram. A $\frac{3}{2}$-Approximation Algorithm for Scheduling Independent Monotonic Malleable Tasks. *SIAM Journ. on Computing*, 37(2):401–412, 2007.
27. T. N'Takpe and F. Suter. Critical Path and Area Based Scheduling of Parallel Task Graphs on Heterogeneous Platforms. In *Proc. of 12th Int. Conf. on Parallel and Distr. Syst. (ICPADS06)*, pages 3–10, Washington, DC, USA, 2006. IEEE Comp. Society.

28. T. N'Takpe, F. Suter, and H. Casanova. A Comparison of Scheduling Approaches for Mixed-Parallel Applications on Heterogeneous Platforms. In *Proc. of 6th Int. Symp. on Parallel and Distr. Computing (ISPDC '07)*, pages 35–42. IEEE Comp. Society, 2007.
29. A. Radulescu, C. Nicolescu, A.J.C. van Gemund, and P. Jonker. CPR: Mixed Task and Data Parallel Scheduling for Distributed Systems. In *Proc. of 15th Int. Parallel & Distr. Processing Symp. (IPDPS01)*, pages 39–48. IEEE Comp. Society, 2001.
30. A. Radulescu and A.J.C. van Gemund. A Low-Cost Approach towards Mixed Task and Data Parallel Scheduling. In *Proc. of 2001 Int. Conf. on Parallel Processing*, pages 69–76. IEEE Comp. Society, 2001.
31. S. Ramaswamy, S. Sapatnekar, and P. Banerjee. A framework for exploiting task and data parallelism on distributed memory multicomputers. *IEEE Trans. Parallel Distr. Syst.*, 8(11):1098–1116, 1997.
32. T. Rauber and G. Rünger. Compiler Support for Task Scheduling in Hierarchical Execution Models. *Journ. Syst. Archit.*, 45(6-7):483–503, 1998.
33. T. Rauber and G. Rünger. Scheduling of Data Parallel Modules for Scientific Computing. In *Proc. of 9th SIAM Conf. on Parallel Processing for Scientific Computing (PPSC)*. SIAM, 1999.
34. T. Rauber and G. Rünger. A Transformation Approach to Derive Efficient Parallel Implementations. *IEEE Transactions on Software Engineering*, 26(4):315–339, 2000.
35. H.J. Sips and K. van Reeuwijk. An Integrated Annotation and Compilation Framework for Task and Data Parallel Programming in Java. In *Parallel Computing: Software Technology, Algorithms, Architectures and Applications, PARCO 2003, Dresden, Germany*, pages 111–118. Elsevier, 2003.
36. T. Tannenbaum, D. Wright, K. Miller, and M. Livny. Condor – a distributed job scheduler. In Thomas Sterling, editor, *Beowulf Cluster Computing with Linux*. MIT Press, October 2001.
37. H. Topcuoglu, S.H., and M.-Y. Wu. Task Scheduling Algorithms for Heterogeneous Processors. In *HCW '99: Proc. of 8th Heterogeneous Computing Workshop*, page 3, Washington, DC, USA, 1999. IEEE Comp. Society.
38. A.J.C. van Gemund. Symbolic performance modeling of parallel systems. *IEEE Trans. Parallel Distrib. Syst.*, 14(2):154–165, 2003.
39. M.Y. Wu and D.D. Gajski. Hypertool: A programming aid for message-passing systems. *IEEE Trans. Parallel Distr. Syst.*, 1(3):330–343, 1990.
40. T. Yang and A. Gerasoulis. Pyrros: Static task scheduling and code generation for message passing multiprocessors. In *6th ACM Int. Conf. on Supercomputing*, pages 428–437, 1992.