Extending the Scheduling Toolkit SEParAT to Support Hybrid Parallel Platforms

Jörg Dümmler and Martin Schulze

Abstract—**Current parallel platforms are increasingly equipped with additional accelerators leading to hybrid system architectures. Parallel applications for these platforms can be implemented using a task-based programming approach. Such an approach facilitates the exploitation of all available execution units including the processor cores and the accelerators. The execution of a task-based application requires scheduling decisions, which may be provided by a suitable scheduling tool.**

This article discusses the extensions of the scheduling toolkit SEParAT to support hybrid cluster architectures. In particular, it first defines the extended programming model for hybrid platforms and the corresponding scheduling problem. The second part of the article describes the integration of this model into SEParAT. A particular focus lies on the extension of SEParAT's input and output interfaces.

Index Terms—**Hybrid architectures, parallel tasks, scheduling, software tool.**

I. Introduction

Many current high performance platforms are equipped with additional accelerators leading to hybrid system architectures. Examples for accelerators are graphics processing units (GPUs) such as the Nvidia Tesla product line or the Intel Many Integrated Core (MIC) architecture including the Intel Xeon Phi.

Complex applications are often composed of multiple program parts where some parts may benefit from an execution on accelerators and other parts may be more suited for an execution on the main processor cores (CPU cores). In this case, a task-based programming approach that decomposes the application into a set of tasks and maps the individual tasks to the most appropriate execution units can help to reduce the execution time of the application. Examples for such applications are linear algebra routines implemented in the MAGMA library [1] like the Hessenberg Reduction [2] and Map-Reduce frameworks designed for hybrid target systems [3]-[5].

The execution of a task-based application requires a schedule that takes the dependencies between the tasks and platform-specific details such as the type and number of available accelerators as well as their computational performance into account. Such a schedule can be obtained dynamically at runtime of the application or statically before the execution of the application. The advantages of the dynamic approach include the support for dynamic task creation at runtime and the availability of dynamic load information when making scheduling decisions. The StarPU system [6] provides runtime support for this approach including dynamic scheduling algorithms. The static approach permits the use of more sophisticated scheduling techniques, which can operate on the entire task graph and provide a runtime estimate for the application that can guide performance optimizations.

Manuscript received October 30, 2014; revised March 11, 2015.

J. Dümmler and M. Schulze are with Technische Universität Chemnitz, Department of Computer Science, 09107 Chemnitz, Germany (e-mail: {djo, schulm}@hrz.tu-chemnitz.de).

This article focuses on the static scheduling for hybrid architectures. In particular, it proposes appropriate extensions to the scheduling framework SEParAT [7], which provides a uniform infrastructure for scheduling algorithms for homogeneous and heterogeneous architectures. The extensions are built atop a programming model for hybrid target architectures that consists of an application model, a platform model, and a corresponding scheduling problem. The application model is based on parallel tasks that can either be executed by an accelerator or by a set of CPU cores. The platform model supports clusters consisting of hybrid compute nodes, which may be equipped with multiple (possibly different) accelerators. The scheduling problem consists of the determination of a feasible schedule that leads to the minimum execution time of a given application on a specific hybrid platform.

SEParAT exhibits a component-based software architecture. The components include user interfaces, input and output components, transformation components for the processing of internal data structures, a generator component to create synthetic scheduling problems, a validation component, and an extensible scheduling algorithm library with support for homogeneous and heterogeneous target architectures. The integration of the programming model for hybrid architectures requires the extension of most of these components. The article especially focuses on the extended input and output components and illustrates the extended interfaces using example specifications. The extensions to SEParAT guarantee backwards compatibility to the already existing model for heterogeneous platforms.

The structure of the article is as follows. Section II gives a short overview of SEParAT. Section III presents the task-based programming model for hybrid platforms that has been incorporated into SEParAT. Section IV describes the extensions to SEParAT in detail. Section V discusses related work and Sect. VI concludes the article.

II. The Scheduling Toolkit SEParAT

This section gives a short overview of the scheduling framework SEParAT (Scheduling Support Environment for Parallel Application Task Graphs) [7], [8] that supports the scheduling of parallel applications in various ways. The main focus of SEParAT lies on *static scheduling* of applications consisting of precedence-constrained *parallel tasks*, i.e., tasks that can be executed by multiple execution units cooperatively. The supported target platforms of SEParAT include homogeneous clusters and heterogeneous clusters-of-clusters, i.e., large heterogeneous clusters that are composed of multiple homogeneous subclusters. The extensions for hybrid platforms are outlined in the following sections.

SEParAT supports two modes of operations:

- Auxiliary tool with command line interface and
- Stand-alone application with graphical user interface (GUI).

The command line interface is mainly intended to facilitate the integration into other programming support tools that require scheduling decisions like, for example, parallelizing compilers or tools that translate parallel specifications into executable code [9]. The GUI assists the user in the evaluation of existing scheduling algorithms and the development of new scheduling algorithms. For example, the GUI can visualize intermediate steps of the scheduling process, internal states and the results of benchmarking runs that can be performed using synthetic scheduling problems and various user-defined parameters.

The main usage scenario of SEParAT is the scheduling of a given application for a particular target platform using a specific scheduling algorithm. In this case, the required input consists of 3 parts: the application structure in form of a *task graph* with annotated cost information, the specification of the parameters of the parallel target platform, and problem-specific parameters like the problem size. The task graph is hierarchical, i.e., each node may consist of a task graph itself, and there may be multiple implementation variants for each parallel task. The cost information are specified using *symbolic runtime formulas* that may depend on problem-specific and platform-specific parameters provided by the other two input files. The output of SEParAT is a schedule that defines the execution order of the parallel tasks, the selected implementation variants, and the assigned execution units of the target platform.

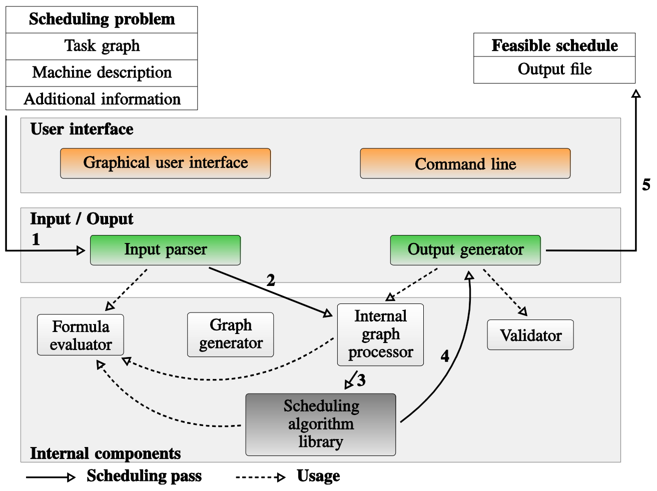


Fig. 1. Software architecture and workflow of SEParAT.

The software architecture of SEParAT is based on components where the core component is an extensible scheduling algorithm library that provides a uniform interface to different scheduling algorithms and includes a plug-in mechanism to add further algorithms. Currently, SEParAT supports 18 scheduling algorithms for homogeneous and 6 algorithms for heterogeneous target platforms.

Fig. 1 gives an overview of the components and shows the workflow of a scheduling pass, which consists of the following 5 steps:

1) The *Input parser* reads the provided input files and creates the corresponding internal data structures.
2) The *Internal graph processor* transforms the given hierarchical task graph into a flat graph and selects appropriate implementation variants for the parallel tasks.
3) The *Scheduling algorithm library* computes a schedule using the algorithm specified by the user.
4) The computed schedule is validated (optional) and postprocessed, i.e., information regarding the hierarchical graph structure and selected implementation variants are added.
5) The *Output generator* creates the output files.

III. Parallel Programming Model for Hybrid Platforms

This section presents the programming model of SEParAT for hybrid architectures that consists of a submodel for the parallel target platform (see Subsect. III.A), a submodel for the parallel application (see Subsect. III.B), the annotation of cost information (see Subsect. III.C), and the corresponding scheduling problem (see Subsect. III.D).

A. Platform Model

The target platform is a heterogeneous distributed memory cluster consisting of c compute nodes $\{N_1, \dots, N_c\}$. Each node may have two types of execution resources: the cores of the central processing unit (CPU) and additional accelerators (called *submachines* in the following), such as graphics processors (GPUs) or the Intel Xeon Phi. This means, the execution resources of node $N_i, i = 1, \dots, c$ encompass

- The nonempty set $\mathfrak{C}_i = \{C_{i,1}, \dots, C_{i,p_i}\}$ of p_i $(p_i > 0)$ CPU cores and
- The set $\mathfrak{Z}_i = \{S_{i,1}, \dots, S_{i,s_i}\}$ of s_i submachines.

Fig. 2(a) shows an example for a hybrid cluster platform consisting of 2 compute nodes.

The CPU cores are assumed to be identical, but the set $\mathfrak{Z}_i$ may include different types of submachines and may also be empty in case a compute node is not equipped with submachines. The compute performance of the execution units is captured by the average execution time of an arithmetical operation, which is denoted by t_i^{C} for the CPU cores of node N_i and by t_i^{CS} for submachine $S_{i,j}$ of node $N_i, i = 1, \dots, c$.

The interconnection network between the execution units of the entire cluster are modeled on two levels. On the lower level, there is the node-internal network that connects all CPU cores and all submachines of the same compute node. This network is assumed to be homogeneous, i.e., data transfers between main memory and individual submachines and data transfers between different submachines are performed at the same speed. To model the performance of the internal network of node $N_i, i = 1, \ldots, c$, we use the startup time t_i^{S} and the byte-transfer time t_i^{B}.

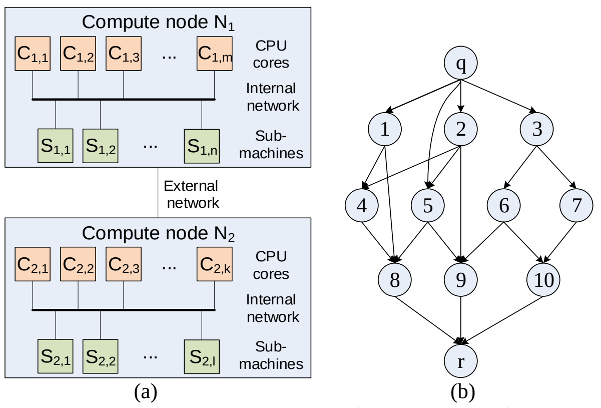


Fig. 2(a) Example for a hybrid cluster platform consisting of 2 compute nodes N_1 and N_2. (b) Example for an application task graph consisting of the entry node q, the exit node r, and the set of parallel tasks $\{1, \ldots, 10\}$.

On the upper level, there is the network that interconnects different compute nodes. This network is assumed to be heterogeneous and the communication performance between nodes N_i and $N_j, 1 \leq i, j \leq c, i \neq j$, is captured by the startup time $t_{i,j}^{\mathrm{S1}}$ on N_i, the startup time $t_{i,j}^{\mathrm{S2}}$ on N_j and the byte-transfer time $t_{i,j}^{\mathrm{B}}$.

B. Application Model

The parallel application consists of a set of parallel tasks with dependencies that are modeled in form of a hierarchical annotated directed acyclic graph $G = (V, E)$. The nodes of the graph include a unique entry node q that represents the input of the application, a unique exit node r that represents the output of the application, and a set of task nodes that represent the parallel tasks of the application. The entry (exit) node is an ancestor (descendant) of all other nodes. An example for an application task graph is shown in Fig. 2(b).

Each parallel task of the application can be either basic or complex. A complex parallel task is represented by an entire directed acyclic graph that describes how the complex task is built up from other parallel tasks. A basic parallel task is not further decomposed and represents the smallest unit visible to SEParAT.

Each parallel task may have multiple implementation variants, where each implementation variant is suited for one or more specific target architectures, i.e., the CPU cores or specific submachines. For example, a parallel task A may have two implementation variants I_1 and I_2 where I_1 may only be executable on CPU cores and the Intel Xeon Phi whereas implementation variant I_2 may only be executable on Nvidia GPUs. As a consequence, parallel task A cannot be executed on Radeon GPUs, since there is no available implementation variant.

The edges of the task graph represent control and data dependencies between the parallel tasks that restrict the possible execution order. A data dependency (A, B) between parallel tasks A and B may lead to communication operations at runtime of the application if A and B are assigned to different execution units, e.g., to different nodes of the cluster or to different submachines of the same node. These communication operations have to be taken into account when determining a suitable schedule for the entire application.

C. Cost Annotations

The nodes and edges of the application task graph are annotated with cost information that provide an estimate of the execution time of the corresponding computation or communication operations depending on the assigned execution units.

The computation costs of the parallel tasks on node $N_i, i = 1, \ldots, c$ of the cluster are captured by two functions

$$T_i^{\mathrm{CPU}}: \quad V \times [1, \ldots, p_i] \to \mathbb{R}^+$$
$$T_i^{\mathrm{SUB}}: \quad V \times [1, \ldots, s_i] \to \mathbb{R}^+$$

where $T_i^{\mathrm{CPU}}(v, p)$ denotes the execution time of parallel task $v \in V$ using p CPU cores of node N_i and $T_i^{\mathrm{SUB}}(v, j)$ denotes the execution time of v on submachine $S_{i,j}$. If a task has multiple implementation variants, the function values of T_i^{CPU} and T_i^{GPU} represent the minimum execution time over all variants on the respective execution units. In case a parallel task cannot be executed on CPU cores or on a specific submachine, the respective function values of T^{CPU} or T^{SUB} are set to infinity. For example, if parallel task v cannot be executed on CPU cores, then $T_i^{\mathrm{CPU}}(v, p) = \infty$ for all nodes $i = 1, \ldots, c$ and all processor numbers $p = 1, \ldots, p_i$.

The unique entry node and the unique exit node have an execution time of zero on each execution unit, i.e.,

$$T_i^{\mathrm{CPU}}(q, p) = T_i^{\mathrm{CPU}}(r, p) = T_i^{\mathrm{SUB}}(q, j) = T_i^{\mathrm{SUB}}(r, j) = 0$$

for each $i = 1, \ldots, c$, all processor numbers $p = 1, \ldots, p_i$ and all submachines $j = 1, \ldots, s_i$.

The costs for the communication operations arising from data dependencies between parallel tasks depend on the amount of data to be transferred and the execution resources assigned to the respective parallel tasks. The set of functions

$$T_{i,j}^{\mathrm{COMM}}: \quad E \times [1, \ldots, p_i] \times \left[1, \ldots, p_j\right] \to \mathbb{R}^+$$

captures these costs where $T_{i,j}^{\mathrm{COMM}}((u, v), p_1, p_2)$ denotes the communication costs arising from edge $(u, v) \in E$ assuming task u is executed on p_1 execution units of cluster node N_i and task v is executed on p_2 execution units of cluster node N_j. We do not distinguish between an execution on a single CPU core and on a single submachine here, since the assumed homogeneous node-internal network leads to identical communication costs for both cases.

D. Scheduling Problem

A schedule $SCHED$ assigns each parallel task $v \in V$ a cluster node N_{NN_v}, a set of execution resources R_v, and a starting point in time $ST_v > 0$. The set R_v is either a subset of the CPU cores of cluster node N_{NN_v}, i.e., $R_v \subseteq \mathfrak{C}_{NN_v}$, or one of the submachines on cluster node N_{NN_v}, i.e., $R_v = \{S_{NN_v,i}\}$ for an $i \in \{1, \ldots, s_{NN_v}\}$. An example for a schedule is shown in Fig. 3.

The execution time T_v^{RUN} of parallel task v is computed depending on the assigned execution resources by

$$T_v^{\mathrm{RUN}} = \begin{cases} T_{NN_v}^{\mathrm{CPU}}(v, |R_v|) & \text{if} \quad R_v \subseteq \mathfrak{C}_{NN_v} \\ T_{NN_v}^{\mathrm{SUB}}(v, i) & \text{if} \quad R_v = \left\{S_{NN_v,i}\right\}. \end{cases}$$

The finish time FT_v of parallel task v is the sum of its starting time and its execution time, i.e.,

$$FT_v = ST_v + T_v^{\mathrm{RUN}}.$$

A schedule is called *feasible*, if it fulfills the following three constraints.

1) Before a parallel task $v \in V$ is started, all predecessors of v must have finished their execution and all required communication operations must have been carried out, i.e., for each edge $(u, v) \in E$ the following inequality has to be fulfilled:

$$ST_v \geq FT_u + T_{i,j}^{\mathrm{COMM}}((u,v), |R_u|, |R_v|)$$

where $i = NN_u$ and $j = NN_v$.

2) Parallel tasks with an overlapping execution time interval have to be executed on disjoint sets of execution units, i.e., for each pair of parallel tasks $(u, v) \in V \times V, u \neq v$, with $[ST_u, FT_u] \cap [ST_v, FT_v] \neq \emptyset$ follows $R_u \cap R_v = \emptyset$.
3) Each parallel task has to be assigned to execution resources that are capable of executing it, i.e., for each parallel task $v \in V$ it is $T_v^{\mathrm{RUN}} \neq \infty$.

The makespan $C_{max}(SCHED)$ of a schedule $SCHED$ denotes the point in time when all computations and communication operations of the entire applications have been terminated. This is achieved at the finish time of the exit node r, i.e., $C_{max}(SCHED) = FT_r$. The scheduling problem is to determine a feasible schedule with a minimum makespan. This is a strongly NP-hard problem as has been shown for the special case of a platform consisting of homogeneous processors and precedence constraints in the form of chains [10].

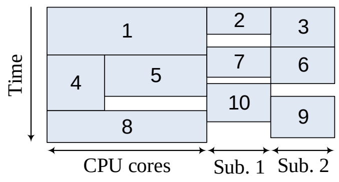
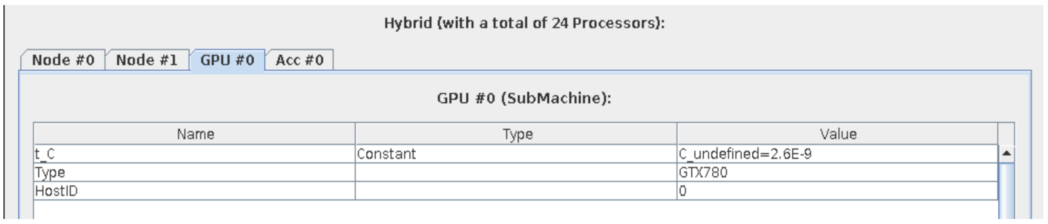


Fig. 3. Example schedule for the application task graph from Fig. 2(b) on a platform consisting of a single node with 2 submachines.

IV. Support for Hybrid Platforms in SEParAT

This section describes the extensions to SEParAT to support hybrid platforms according to the programming model presented in Sect. III. An overview of all extensions is given in Subsect. IV.I.A and Subsect. IV.B discusses the extended input and output interfaces.



Hybrid (with a total of 24 Processors):

GPU #0 (SubMachine):

Name	Type	Value
t_C	Constant	C_undefined=2.6E-9
Type		GTX780
HostID		0

Fig. 4. Properties of a hybrid parallel platform displayed by the graphical user interface of SEParAT.

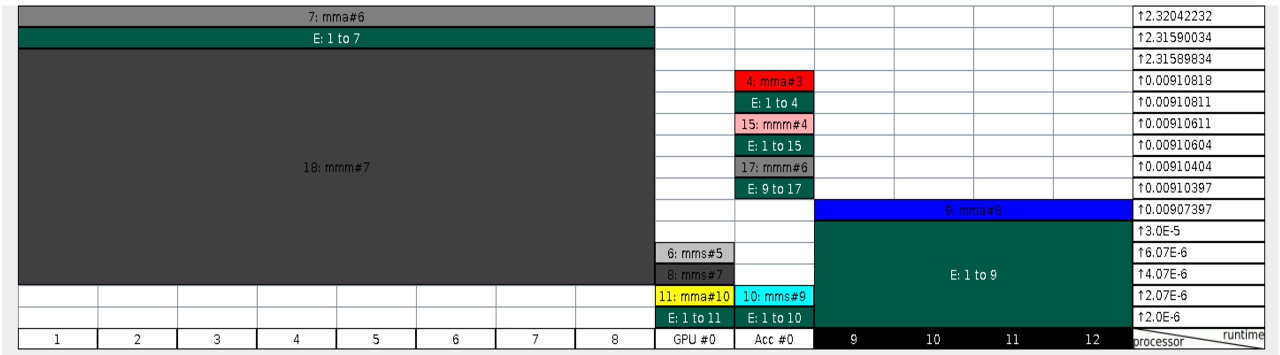


Fig. 5. Fragment of a schedule shown by the graphical user interface of SEParAT. The green boxes illustrate communication operations resulting from data dependencies and the colored boxes symbolize the execution of parallel tasks.

A. Overview of the Extensions for Hybrid Platforms

To support scheduling for hybrid platforms, several components of SEParAT had to be extended, see Fig. 1 for an overview of the software structure. In particular, the following extensions have been made.

- The input and output interfaces are modified to account for the additional submachines of hybrid platforms and the additional cost information for the different architectures. The modified interfaces require extensions to the *Input parser* and *Output generator* components of SEParAT, see Subsect. IV.B for details.
- The internal interfaces and data structures have been extended according to the modified input and output interfaces.
- Specifically adapted scheduling algorithms have to be included in the *Scheduling algorithm library*. Currently, there exists a simple list scheduler that assigns parallel tasks to the next free execution unit that can execute the task. This can either be a submachine or all available processors of a compute node. Further scheduling algorithms will be added in the future.
- The *Graph generator* that is used to create synthetic scheduling problems for benchmarking scheduling algorithms is extended to create cost information for hybrid platforms. Furthermore, this component now also

supports the generation of synthetic hybrid platform configurations based on user-provided parameters like the average number of submachines per cluster node and a heterogeneity factor where a low factor produces rather similar cluster nodes (same type and number of submachines) and a high factor produces completely different cluster node configurations.

- The illustration of scheduling problems and generated schedules for hybrid platforms is incorporated into the *Graphical user interface*. Fig. 4 shows the properties of a hybrid platform and Fig. 5 shows a computed schedule in the GUI.

B. Extensions of the Input and Output Interfaces

The input of SEParAT consists of a platform specification (see Subsect. IV.B.1), a specification of the application task graph (see Subsect. IV.B.2), and a definition of application specific parameters like the problem size. The output of SEParAT is a feasible schedule for the provided input scheduling problem (see Subsect. IV.B.3).

1) Specification of hybrid platforms

The input specification for hybrid platforms has been designed to be backward compatible to the specification of heterogeneous cluster-of-cluster platforms in SEParAT. An example for a hybrid cluster specification is given in Fig. 6. A complete specification consists of a list of compute nodes (lines 3–24) and the communication performance of the interconnection network between the compute nodes, which is defined by the network startup times t^{S1} and t^{S2} and the network byte-transfer time t^{B} (lines 27–33).

```
1  <HeterogeneousMachineDesc Name="Hybrid">
2   <!-- Specification of the cluster nodes -->
3   <Machines>
4    <!-- Cluster Node 0 -->
5    <Machine Id="0" Name="Node #0"
6                   Processors="8">
7     <Constant Name="t S" Value="2.0E-6"/>
8     <Constant Name="t B" Value="1.2E-9"/>
9     <Constant Name="t C" Value="6.9E-8"/>
10    <SubMachine Id="1" Name="GPU #0"
11                       Type="GTX780">
12     <Constant Name="t C" Value="2.6E-9"/>
13    </SubMachine>
14    <SubMachine Id="2" Name="Acc #0"
15                             Type="IntelPhi">
16     <Constant Name="t C" Value="4.9E-9"/>
17    </SubMachine>
18   </Machine>
19   <!-- Cluster Node 1 -->
20   <Machine Id="1" Name="Node #1"
21                       Processors="16">
22     <!-- [...] -->
23   </Machine>
24  </Machines>
25
26  <!-- Interconnection between nodes -->
27  <MachineConnections>
28   <Connection Endpoint1="0" Endpoint2="1">
29    <Constant Name="t S1" Value="3.0E-05"/>
30    <Constant Name="t_S2" Value="3.0E-05"/>
31    <Constant Name="t_B"  Value="1.1E-09"/>
32   </Connection>
33  </MachineConnections>
34 </HeterogeneousMachineDesc>
```

Fig. 6. Example for a hybrid cluster platform specification.

A single cluster node is defined by the number of CPU cores (line 6), the communication performance of the node-internal interconnection network (lines 7–8) and the compute performance of the processor cores (line 9). Additionally, each cluster node contains a list of zero or more submachines (lines 10–17) where each submachine has a unique identifier and a submachine type, which is used to determine whether a given parallel task is executable on this submachine. The compute performance of a submachine is specified by the average execution time of an arithmetic instruction t^{C} (lines 12 and 16).

For the incorporation of other cost models, the platform specification also supports additional user-defined constants and functions in the definition of cluster nodes or submachines like, for example, the execution time of a broadcast operation depending on the number of participating CPU cores. These constants and functions may then be used in the runtime formulas of the parallel tasks in the application specification.

```
<!-- definition of external parameters -->
<!-- problem size -->
<ProblemParam Name="n" DefaultValue="1024"/>
<!-- compute power -->
<MachineParam Name="t_C"/>

<!-- data types and data distrib types -->
<DataType Name="myMatrix" DataType="matrix"
  C-Type="double" Dimension="2" Size="n;n">
  <DataDistrib Name="block"
      Description="BLOCK"/>
</DataType>

<!-- basic parallel task definition -->
<Module Name="myNode" Id="1">
  <Param Name="in"  Id="1" Type="myMatrix"/>
  <Param Name="out" Id="2" Type="myMatrix"/>
  <Implementation Name="mod1_block" Id="1">
    <Distrib ParamRef="1" Type="block"/>
    <Distrib ParamRef="2" Type="block"/>
    <BasicModule>
      <Runtime Formula="T_par(p,n,t_C)=
         0.1*t_C*n^2+(0.9*t_C*n^2)/p"/>
      <Runtime Type="GTX780" Formula=
          "T_par(n,t_C)=0.05*t_C*n^2"/>
    </BasicModule>
  </Implementation>
</Module>

<!-- complex parallel task definition -->
<Module Name="task graph" Id="2">
  <Param Name="in"  Id="1" Type="myMatrix"/>
  <Param Name="out" Id="2" Type="myMatrix"/>
  <Implementation Name="main impl" Id="1">
    <Distrib ParamRef="1" Type="block"/>
    <Distrib ParamRef="2" Type="block"/>
    <ComplexModule>
      <!-- nodes of the task graph -->
      <StartNode Name="entry" Id="1"/>
      <Node Name="n1" Id="2" ModuleRef="1"/>
      <StopNode Name="exit" Id="3"/>
      <!-- edges of the task graph -->
      <Edge Id="1" SourceNodeId="1"
         SourceParamId="1" TargetNodeId="2"
         TargetParamId="1"/>
      <Edge Id="2" SourceNodeId="2"
         SourceParamId="2" TargetNodeId="3"
         TargetParamId="2"/>
    </ComplexModule>
  </Implementation>
</Module>

<!-- definition of the application root -->
<MainModule ModuleRef="2"/>
```

Fig. 7. Example for a specification of a parallel application.

2) *Application specification*

The specification of the application task graph for SEParAT has been extended to include cost information for different types of submachines. An example specification is shown in Fig. 7. It consists of 4 parts:

1) The definition of external parameters that are provided as part of the platform specification or problem description (lines 3–5);
2) Definitions of the data types and data distribution types used within the specification (lines 8–12);
3) Definitions of basic and complex parallel tasks (lines 15–51);
4) The indication of a specific complex parallel task that represents the entire application (line 54).

The specification of a parallel task consists of a set of input and output parameters with a corresponding data type (lines 16–17, 32–33) and a list of implementation variants. Each implementation variant defines the data distributions for the input and output parameters (lines 19–20, 35–36) and either defines a runtime prediction (basic parallel tasks that are not further decomposed) or includes an entire task graph (complex parallel tasks). In the case of a basic parallel task, a set of *symbolic runtime formulas* is given (lines 22–25). Each runtime formula has a type attribute, which defines the target architecture (CPU cores or a specific type of submachine). To guarantee backward compatibility, a missing type attribute refers to CPU cores. If there is no runtime formula for a specific architecture, SEParAT assumes that this basic task cannot be executed on this type of submachine. For example, the basic parallel task in lines 15–28 can be executed on CPU cores and on submachines of type *GTX780*, but not on submachines with type *IntelPhi*.

The task graph specification of a complex parallel task consists of a set of nodes (lines 39–41) and a set of edges (lines 43–48). Each node (except the unique start and stop node) refers to one of the parallel tasks in the application specification and each edge connects an output parameter of the source parallel task with an input parameter of the target parallel task.

3) *Schedule specification*

The output format for schedules produced by SEParAT has been extended to account for the execution of parallel tasks on the additional submachines of the target platform. Fig. 8 shows an example schedule for a hybrid platform. A schedule consists of communication operations (lines 2–12) and the execution of parallel tasks (lines 13–19) where each operation has an associated start time and an associated finish time. A communication operation additionally contains the identifiers of the source and target nodes in the underlying task graph and the definition of the source and target execution units. The execution of a parallel task includes the identifier of the parallel task, the selected implementation variant and the assigned execution units.

The execution units can be either a set of CPU cores or a specific submachine. In the former case, a list of the global core numbers is included where the cores are numbered consecutively over all compute nodes of the target platform (line 6). In the latter case, the global identifier of the submachine is given where the global identifier is obtained by consecutively numbering all submachines of the platform (line 10).

```
1  <Schedule Id="2" Makespan="0.218">
2    <DataRedistribution Id="1"
3      SourceNodeId="1" TargetNodeId="2"
4      StartTime="0.0" FinishTime="0.00374">
5      <SourceMachine>
6        <ProcessorGroup>1 2 3 4 5 6 7 8
7        </ProcessorGroup>
8      </SourceMachine>
9      <TargetMachine>
10       <SubMachine>0</SubMachine>
11     </TargetMachine>
12   </DataRedistribution>
13   <ModuleCall Id="2" Name="n1"
14     ModuleRef="1" ImplementationRef="1"
15     StartTime="0.00374" FinishTime="0.218">
16    <TargetMachine>
17     <SubMachine>0</SubMachine>
18    </TargetMachine>
19   </ModuleCall>
20 </Schedule>
```

Fig. 8. Example for a schedule produced by SEParAT.

V. Related Work

The execution of a parallel application on a hybrid platform can either be based on a data parallel or a task parallel approach. In the data parallel case, the input data is partitioned over the available execution units and processed according to the owner-computes rule. Examples for such implementations are the Jacobi method [11], the FFT [12], and the determination of connected components in graphs [13]. The partitioning can be based on profiling information [14], [15] or on static source code analysis in a compiler [16].

The task parallel approach decomposes the application into a set of tasks where each task is assigned to one or more execution units. This assignment can be performed statically or dynamically at runtime. The static approach has been used, for example, for many hybrid implementations of linear algebra routines in the MAGMA library [1], the Hessenberg Reduction [2], [11], or Continuous Collision Detection [12].

The dynamic approach has been used for Map-Reduce frameworks [3]-[5] and for image processing [17]. This implementation approach is supported by runtime systems like StarPU [6], Harmony [20] and Merge [21], and dynamic scheduling algorithms [22], [23]. In contrast to these approaches, SEParAT offers tool support for the static scheduling, which is especially beneficial for regular applications with a static task structure.

VI. Conclusion

This article has presented the extensions made to the scheduling framework SEParAT to support hybrid cluster platforms. The underlying platform model supports arbitrary heterogeneous cluster systems where each cluster node can be equipped with multiple accelerators. The application model of SEParAT assumes a hierarchical task-based application with precedence constraints between the parallel tasks, which can be either executed on accelerators or on a subset of the processor cores of a cluster node. The support for hybrid platforms requires the extensions of the internal components of SEParAT. The article has shown examples for extended input and output specifications.

Future work encompasses the development and implementation of scheduling algorithms for the extended programming model in SEParAT and the evaluation of these algorithms using different applications from scientific computing.

REFERENCES

[1] MAGMA Version 1.5. (2014). [Online]. Available: http://icl.cs.utk.edu/magma

[2] S. Tomov, J. Dongarra, and M. Baboulin, "Towards dense linear algebra for hybrid GPU accelerated manycore systems," *Parallel Computing*, vol. 36, no. 5-6, pp. 232-240, 2010.

[3] W. Jiang and G. Agrawal, "MATE-CG: A Map Reduce-Like Framework for Accelerating Data-Intensive Computations on Heterogeneous Clusters," in *Proc. the 26th IEEE Int. Parallel Distributed Processing Symposium (IPDPS'12)*, IEEE, 2012, pp. 644-655.

[4] K. Shirahata, H. Sato, and S. Matsuoka, "Hybrid map task scheduling for GPU-based heterogeneous clusters," in *Proc. the 2nd IEEE Int. Conf. on Cloud Computing Technology and Science (CLOUDCOM'10)*, Washington, DC, USA: IEEE Computer Society, 2010, pp. 733-740.

[5] L. Chen, X. Huo, and G. Agrawal, "Accelerating MapReduce on a coupled CPU-GPU architecture," in *Proc. the Int. Conf. on High Performance Computing, Networking, Storage and Analysis (SC'12)*, Los Alamitos, CA, USA: IEEE Computer Society, 2012.

[6] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "StarPU: A unified platform for task scheduling on heterogeneous multicore architectures," *Concurrency and Computation: Practice and Experience*, vol. 23, no. 2, pp. 187-198, 2011.

[7] J. Dümmler, R. Kunis, and G. Rünger, "SEParAT: Scheduling Support Environment for Parallel Application Task Graphs," *Cluster Computing*, vol. 15, no. 3, pp. 223-238, 2012.

[8] SEParAT project web site. [Online]. Available: http://www.tu-chemnitz.de/informatik/PI/forschung/projekte/genMTS

[9] J. Dümmler, T. Rauber, and G. Rünger, "Programming support and scheduling for communicating parallel tasks," *Journal of Parallel and Distributed Computing*, vol. 73, no. 2, pp. 220-234, 2013.

[10] J. Du and J.-T. Leung, "Complexity of scheduling parallel task systems," *SIAM Journal on Discrete Mathematics*, vol. 2, no. 4, pp. 473-487, 1989.

[11] S. Venkatasubramanian and R. W. Vuduc, "Tuned and wildly asynchronous stencil kernels for hybrid CPU/GPU systems," in *Proc. the 23rd Int. Conf. on Supercomputing (ICS'09)*, Yorktown Heights, NY, USA: ACM, 2009, pp. 244-255.

[12] Y. Ogata, T. Endo, N. Maruyama, and S. Matsuoka, "An efficient, model-based CPU-GPU heterogeneous FFT library," in *Proc. the 22nd IEEE Int. Symp. on Parallel and Distributed Processing (IPDPS'08)*, IEEE, 2008.

[13] D.S. Banerjee and K. Kothapalli, "Hybrid algorithms for list ranking and graph connected components," in *Proc. the 18th Int. Conf. on High Performance Computing (HiPC'11)*, Washington, DC, USA: IEEE Computer Society, 2011.

[14] C.-K. Luk, S. Hong, and H. Kim, "Qilin: Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping," in *Proc. the 42nd Annual IEEE/ACM Int. Symp. on Microarchitecture (MICRO'09)*, New York, NY, USA: ACM, 2009, pp. 45-55.

[15] V.T. Ravi and G. Agrawal, "A dynamic scheduling framework for emerging heterogeneous systems," in *Proc. the 18th Int. Conf. on High Performance Computing (HIPC'11)*, Washington, DC, USA: IEEE Computer Society, 2011.

[16] D. Grewe and M. O'Boyle, "A static task partitioning approach for heterogeneous systems using OpenCL," *Compiler Construction, Lecture Notes in Computer Science*, vol. 6601, pp. 286-305, 2011.

[17] J. Muramatsu, T. Fukaya, S.-L. Zhang, K. Kimura, and Y. Yamamoto, "Acceleration of Hessenberg Reduction for nonsymmetric eigenvalue problems in a hybrid CPU-GPU computing environment," *Int. Journal of Networking and Computing*, vol. 1, no. 2, pp. 132-143, 2011.

[18] D. Kim, J.-P. Heo, J. Huh, J. Kim, and S.-E. Yoon, "HPCCD: Hybrid parallel continuous collision detection using CPUs and GPUs," *Computer Graphics Forum*, vol. 28, no. 7, pp. 1791-1800, 2009.

[19] G. Teodoro, T. M. Kurc, T. Pan, L. A. D. Cooper, J. Kong, P. Widener, and J. H. Saltz, "Accelerating large scale image analyses on parallel, CPU-GPU equipped systems," in *Proc. the 26th IEEE Int. Symp. on Parallel and Distributed Processing (IPDPS'12)*, 2012, pp. 1093-1104.

[20] G. F. Diamos and S. Yalamanchili, "Harmony: An execution model and runtime for heterogeneous many core systems," in *Proc. the 17th Int. Symp. on High Performance Distributed Computing (HPDC'08)*, New York, 2008, pp. 197-200.

[21] M. D. Linderman, J. D. Collins, H. Wang, and T. H. Meng "Merge: A programming model for heterogeneous multi-core systems," *SIGPLAN Notices*, vol. 43, no. 3, pp. 287-296, 2008.

[22] A. P. D. Binotto, B. M. V. Pedras, M. Götz, A. Kuijper, C. E. Pereira, A. Stork, and D. W. Fellner, "Effective dynamic scheduling on heterogeneous multi/manycore desktop platforms," in *Proc. the 22nd Int. Symp. on Computer Architecture and High Performance Computing Workshops (SBAC-PADW'10)*, 2010, pp. 37-42.

[23] M. Becchi, S. Byna, S. Cadambi, and S. Chakradhar, "Data-aware scheduling of legacy kernels on heterogeneous platforms with distributed memory," in *Proc. the 22nd ACM Symp. on Parallelism in Algorithms and Architectures (SPAA'10)*, 2010, pp. 82-91.

Jörg Dümmler received a doctoral degree in computer science from Technische Universität Chemnitz, Germany in 2010. Since then, he has been working as a postdoctoral researcher at this institution. His current research interests include high-level parallel programming models, mixed and hybrid parallel algorithms, scheduling for parallel applications, and tool support for scientific programming.

Martin Schulze received the diploma degree from Technische Universität Chemnitz, Germany, in 2014. His research interests include optimization and scheduling.