Lookahead Scheduling for Reconfigurable GRID Systems

Jesper Andersson[1], Morgan Ericsson[1], Welf Löwe[1], and Wolf Zimmermann[2]

[1] Software Technology Group, MSI, Växjö universitet
{jesan,mogge,wlo}@msi.vxu.se
[2] Martin-Luther-Universität Halle-Wittenberg, Institut für Informatik,
D-06099 Halle/Saale, Germany
zimmer@informatik.uni-halle.de

Abstract. This paper proposes an approach to continuously optimizing parallel scientific applications with dynamically changing architectures. We achieve this by combining a dynamic architecture and lookahead malleable task graph scheduling.

1 Introduction

There is a rich body of research on both off-line scheduling and on-line load balancing. The former produce good results for statically know programs, especially if the communication behavior is know statically. The latter is commonly used when programs or hardware configurations change during execution. The disadvantage of load balancing is its unawareness of the applications, leading to limited anticipation of future load situations. Re-scheduling of changed applications or hardware configurations is a non option since the algorithms are too slow.

This paper contributes by making off-line scheduling algorithms applicable for dynamic scenarios. We approach the problem by defining a Dynamic Service Architecture (DSA), responsible for managing event-driven changes to applications and continuously generating schedules for possible future situations. In order to make the continuous generation of schedules feasible we apply malleable task graph scheduling. This allows for faster scheduling by the reuse of partial schedules.

The paper is organized as follows. Section 2 gives the foundations for the execution and programming model, and introduces the DSA. Section 3 defines the mapping between the two architectural levels of the DSA. This section introduces the composition and the scheduling models. Finally, Section 4 concludes the paper.

2 Application Scenario

This section introduces the foundations of the programming and execution model of the intended system and defines the Dynamic Service Architecture.

M. Danelutto, D. Laforenza, M. Vanneschi (Eds.): Euro-Par 2004, LNCS 3149, pp. 263–270, 2004.

2.1 Programming and Execution Model

The hardware infrastructure, *GRID System* in the following, consist of: Sensors called *input nodes*, generating a stream of *input values*, computation processors (*nodes*) and an interconnecting network for communication. On this GRID system *applications* are executed processing data from sensors. These applications are data parallel programs. Their inputs are sensor input values directly or the output of other applications. If the input of application a is the output of an application a', a is called *data-dependent* on a'. This is denoted by $a' \rightarrow a$. Applications are stateless, data driven, functions.

For individual applications we assume a High Performance Fortran (HPF)-like programming model, with data parallel synchronous programs without any data distribution. We can model such an application by a family of taskgraphs $G_x = (V_x, E_x, \tau_x)$. Scientific applications can automatically be compiled to such a family of task-graphs [5]. The tasks $v \in V_x$ model local computations without access to shared memory. $\tau(v)$ is the execution time of task v on the target machine, and there is a directed edge from v to w iff v writes a value to shared memory, that is read later by task w. Task-graphs are always acyclic. G_x does not always depend on the actual input x. In many cases of practical relevance, it only depends on the problem size n. We call these program *oblivious* and denote their task graphs by G_n. We write G instead of G_n if n is arbitrary but fixed. The *height* of a task v, denoted by $h(v)$, is the length of the longest path from a task with in-degree 0, to the task v.

The hardware is modeled by the LogP [2] model: in addition to the computation costs τ, it models communication costs with parameters Latency, $o$verhead, and $g$ap (which is actually the inverse of the bandwidth per processor). In addition to L, o, and g, parameter P describes the number of processors. Moreover, there is a capacity constraint: at most $\lceil L/g \rceil$ messages are in transmission in the network from any processor to any processor at any time. A send operation that exceeds this constraint stalls.

A LogP-schedule is a schedule that obeys the precedence constraints given by the task-graph and the constraints imposed by the LogP-machine, i.e., sending and receiving a message takes time o, there must be at least time g between two consequential send or receive operations, there must be at least time L between the end of a send task and the beginning of the corresponding receive task, and the capacity constraint must be obeyed. $TIME(s)$ denotes the execution time of schedule s, i.e., the time at which the last task finishes.

The *configuration* of a *system* is defined by: (i) its set of applications, (ii) the data dependencies, and (iii) Quality of Service (QoS) parameters for the different applications.

2.2 Dynamic Service Architecture

A configuration might change over time, i.e. it is dynamic. These changes are *user-*, *application-*, or *system-triggered*. The Dynamic Service Architecture (DSA) manages change the running system. The DSA can be seen as a conceptual control system listening to events from and reconfiguring a physical processing accordingly. The architecture of the processing system is defined by a composition of data-parallel applications. The focus of reconfiguration is, for this paper, the rescheduling.

The user in our scenario controls a certain set of applications. A typical user-triggered change is adding an application to the system or removing it after gaining the results. Some applications act as detectors, recognizing certain patterns in the processed data, that will require a reconfiguration. Application-triggered change is the detection of an interesting sensor activity requiring changed QoS parameters. The complexity of applications might be input dependent. Certain inputs may lead to load peaks in applications. In order to guarantee the required quality of service to applications, certain others applications are postponed to off-line computations. This is a typical system-triggered change.

The DSA use two architecture levels. A scheduler establishes a one-to-one mapping M between the *conceptual* processing architectures, A, and the actual *physical* running systems $M : A \rightarrow I$. User-triggered changes occur on the conceptual level. As a result a new conceptual system-architecture $a \in A$ is activated. This triggers computation of a corresponding implementation $i = M(a)$, including the computation of a new schedule that is distributed to the physical level. Inversely, application- and system-triggered changes occur on the physical level. The level should reflect the situation of the level in order to handle subsequent user-triggered changes in a correct manner. Hence, application- and system-triggered changes also affect the conceptual. However, we distinguish between conceptual events implementing user-triggered change requests and physical events implementing application- and system-triggered events. Both event-classes initiates the generation of new implementations using the translation and scheduling $M : A \rightarrow I$.

3 Mapping Conceptual to Physical Architecture

This section continues the discussion of the reconfigurations performed by the DSA, more specifically the mapping from a conceptual to a physical implementation. This mapping consists of the composition of applications to systems, discussed in Section 3.1 and scheduling. The scheduling is implemented as a continuous process, generating all possible schedules for a given state. This *lookahead* scheduling is defined in Section 3.2. The *lookahead*-scheduling must be as fast as possible, since it defines the rate of change events that can be dealt with. This means that we need to speed-up the classic static scheduling algorithms. This is accomplished using malleable task scheduling, which is described in Section 3.3.

3.1 Static Composition of Applications to Systems

So far, we defined the components: data-parallel applications, translated to task graphs and scheduled to the infrastructure. For each component, we can determine an upper time bound for its execution. Each component implements a function mapping an input array a_i to an output array a_o.

To this end, composition of components can be done by defining a program using these components and assigning the output array of one component to the input of another. Obviously, this system is a data-parallel program, too. It can be compiled and scheduled just like the individual components.

The computation of an optimum LogP-schedule is known to be NP-hard. However, good approximations and heuristics, c.f. our own contributions in task scheduling, e.g. [3], guarantee a small constant factor in delay. In practice, the results are even closer to the optimum. Moreover, an upper time bound for $TIME(s)$, the execution time of a schedule s, can be determined statically.

Adding/removing a component requires a complete re-translation of the system to a new task graph and a rescheduling of the new task-graph. This is, in principle, not a problem since adding/removing a component can be planned and prepared offline. After the complete schedule is computed for the new system, the computation can replace the running system with the new system. However, for application- and system-triggered changes the delay to prepare for a change is not acceptable. Therefore, we compute new schedules before they are actually needed. We call this *lookahead scheduling*.

3.2 Lookahead Scheduling

As mentioned before, we distinguish between conceptual events E_c implementing user-triggered change requests and physical events E_p implementing application- and system-triggered events. If an $e_c \in E_c$ occurs, we compute the new architecture $a' \in A$ of the processing system, map it to a implementation architecture $i' = M(a')$, compute the delta between the current implementation architecture i and i and deploy that delta to the GRID system. If a $e_p \in E_p$ occurs, the reconfiguration must be finished in milliseconds. However, we exploit the assumption that the expectation on the rate of such events E_p is rather low. The fundamental principle for the optimization of our dynamic reconfiguration is to employ a continuous implementation mapping with a *lookahead schema*. For each system architecture A, we pre-compute possible changes Δ w.r.t. possible system events E_p. More specifically: given a current (baseline) architecture $a \in A$ and each possible system event $e_p \in E_p$, we compute the evolved architecture $a' = \Delta(e_p, a)$. After having determined possible deltas, we have a set of *lookahead(1)* architectures. These are mapped in the same way as the base-line-architecture is mapped to *lookahead(1)*- implementations: $i' = M(a')$. Together with the current baseline implementation $i = M(a)$, these possible *lookahead(1)*-implementations are deployed. If this process is not interrupted by other change events, we can react on events to come very swiftly. Details for both scenarios are defined in [1].

The frequency at which we can tolerate change events is the inverse of the delay for computing the lookahead schedules. This delay can be reduced by two means: (i) performing composition on task graph level instead of application level and (ii) using predefined schedules for the task graphs. Both will be discussed below.

Instead of composing data-parallel applications to a data-parallel system, which is then translated to task graphs and scheduled to the infrastructure, we bookkeep the task graphs of the individual applications and just compose these task graphs. Only a new application is translated into a new task graph. Inversely, removal leads to disconnecting the corresponding task graphs and deleting transitively depending tasks.

While reusing task graphs is straight forward, reusing schedules is not, since an optimum schedule (or its approximation) does not necessarily keep the schedules for the different task graphs distinct. Instead, it might merge tasks of different task graphs into

one process. Moreover, optimum schedules of individual task graphs (or their approximations) are, in general, not part of the optimal schedule for a composed system (or its approximation).

This problem is approached by modeling task graphs as *malleable tasks* and systems with *malleable task graphs*. A malleable task is a task that can be executed on $p = 1 \ldots P$ processors. Its execution time is described by a non-increasing function τ of the number of processors p actually used. For each task graph the schedules s_p can be pre-computed for $p = 1 \ldots P$ and $\tau(p) = TIME(s_p)$. A malleable task graph is recursively defined as a task graph over malleable tasks, i.e. nodes are ordinary task graphs or malleable task graphs and edges are the data-dependencies between them.

3.3 Scheduling Malleable Task Graphs

We now show how malleable task-graphs stemming from oblivious programs can be scheduled. The basic idea is to schedule a malleable task-graph layer by layer. By definition, a layer contains only independent tasks. Hence, the results of [4] can be applied when scheduling a layer. After scheduling a layer one builds a communication phase. In order to determine the weight function of a malleable task v, a malleable task graph within v is scheduled for $p = 1, \ldots, P$ processors. If the task-graph only contains atomic tasks then traditional algorithms are applied. The following algorithm implements these ideas:

Algorithm $schedule(G, P)$
INPUT: Malleable Task Graph $G = (V, E)$ with Layers $\Lambda_0, \ldots, \Lambda_m$
Number of available processors P
OUTPUT: A schedule s for G
if each $v \in V$ is atomic **then**
determine the weights τ_v;
compute a schedule s of G; //any traditional scheduling algorithm suffices
return s;
end;
for $i := 0, \ldots, m$ **do**
for each $v \in \Lambda_i$ **do**
if v is malleable **then**
let G_v be the malleable task graph contained in v;
for $j := 1, \ldots, p$ **do**
$s_v(j) := schedule(G_v, j)$;
$T_v(j) := \min(makespan(s_v(j)), T_v(j-1)$; //$T_v(0) = \infty$
endfor;
else determine τ_v;
for $j := 1, \ldots, p$ **do** $T_v(j) := \tau_v$;
endif;
endfor;
compute a schedule s_i for the tasks in Λ_i using [4];
schedule the communication from the schedule s to s_i;
extend s by this communication and s_i;
endfor;
return s;
end $schedule$

The computation of the schedules in the malleable tasks need only be done once.

In order to analyze the make-span of schedules computed by the algorithm, the time required for a communication phase, the make-span of schedules for task-graphs with atomic tasks, and the make-span of scheduling independent malleable tasks has to be determined. The worst case for a communication phase is an all-to-all communication. One could extend the underlying machine model with such communications or use a scheduling approach for implementing such a communication phase (e.g the approach of [3] for the LogP-model). In this article it is sufficient that a communication phase for p processors costs time $\tau_{comm}(n,p)$ where n is the total amount of data communicated. For the purpose of this article it is just sufficient to assume that the make-span of a schedule for a task-graph G containing only atomic tasks can be estimated by an approximation factor c, i.e. the make-span is at most $c \cdot T_{\mathsf{opt}}(G,p)$ where $T_{\mathsf{opt}}(G,p)$ is the make-span of an optimal schedule for G on p processors. For scheduling n independent tasks Λ onto p processors, we use the results of [4], i.e., any schedule has a make-span of at most $\sqrt{3} \cdot T_{\mathsf{opt}}(\Lambda(p))$. A better approach will reduce the approximation factor.

We define a degree of malleability for the analysis of the make-span. Since the number of hierarchy levels play a role, we inductively define the *hierarchy level* of a malleable task $v \in V$ and the hierarchy level of a malleable task-graph as follows:

- If v is not malleable the hierarchy level of v is 0
- If v is a malleable task-graph $G = (V,E,T)$ then the hierarchy level of v is that of G
- The hierarchy level of a malleable task graph $G = (V,E,T)$ containing only atomic tasks is 0
- The hierarchy level of a malleable task graph $G = (V,E,T)$ containing at least a malleable task is $k+1$ where k is the maximal hierarchy level of a task $v \in V$

The *work* of a malleable task graph $G = (V,E,T)$ is defined as $W(G) \triangleq \sum_{v \in V} \tau_v(1)$. The work of a set of tasks $V' \subseteq V$ is defined as $W(V') \triangleq \sum_{v \in V'} \tau_v(1)$.

The *degree of malleability*, $\mu(G,P)$ of a malleable task graph $G = (V,E,T)$ with layers $\Lambda_0, \ldots, \Lambda_n$ is inductively defined as follows:

i. $\mu(G,p) \triangleq 1$ iff each $v \in V$ is atomic

ii. $\mu(\Lambda_i,p) \triangleq \min\left(\min_{w \in \Lambda_i} \mu(w,p), \dfrac{W(\Lambda_i)/p}{W(\Lambda_i)/p + \tau_{comm}(p)}\right)$ where $\mu(w,p) = 1$ if w is atomic.

iii. $\mu(G,p) \triangleq \min_{i=0}^{n-1} \mu(\Lambda_i,p)$

Theorem 1. *Let $G = (V,E,T)$ be a malleable task-graph with hierarchy level k and s be a schedule for P processors computed by the above approach. Then the make-span for the schedule s is at most: $TIME(s) \leq c \cdot 3^{k/2} \frac{T_{\mathsf{opt}}(G)}{\mu(G,P)}$ where a scheduling algorithm with approximation factor c is used for each malleable task in any level of the hierarchy which has only atomic tasks.*

Proof. For $k = 0$, the claim states that $TIME(s) \leq c \cdot T_{\mathsf{opt}}(G)$ since $\mu(G,P) = 1$ in this case. The claim holds since G only contains atomic tasks and G is scheduled by

an algorithm guaranteeing an approximation factor c. For $k \geq 1$, we prove the slightly stronger claim that $TIME(s) \leq c \cdot 3^{k/2} \cdot T_{\mathsf{opt}}(G)/\mu(G,P) - \tau_{comm}(P)$ by induction on k.

CASE $k = 1$: We have to show that $TIME(s) \leq c \cdot \sqrt{3} \cdot T_{\mathsf{opt}}(G)/\mu(G,P) - \tau_{comm}(P)$.

Let $TIME(s_i)$ the make-span of the schedule s_i for layer Λ_i. Observe that for the case $k = 1$, any $v \in V$ is malleable task that is a task-graph containing only atomic tasks. Thus, for any of these tasks x it holds that they have a schedule s_x with $TIME(s_x) \leq c \cdot T_{\mathsf{opt}}(x)$. Hence, the layer Λ_i, using the result of [4], it holds $TIME(s_i) \leq c \cdot \sqrt{3} \cdot T_{\mathsf{opt}}(\Lambda_i)$ where $T_{\mathsf{opt}}(\Lambda_i)$ is the make-span of an optimal schedule for Λ_i. It holds:

$$
\begin{aligned}
TIME(s) &\leq \sum_{i=0}^{n-1} (TIME(s_i) + \tau_{comm}(P)) + TIME(s_n) \\
&= \sum_{i=0}^{n} (TIME(s_i) + \tau_{comm}(P)) - \tau_{comm}(P) \\
&\leq \sum_{i=0}^{n} (c \cdot \sqrt{3} \cdot T_{opt}(\Lambda_i) + \tau_{comm}(P)) - \tau_{comm}(P) \text{ (see above)} \\
&\leq c \cdot \sqrt{3} \cdot \sum_{i=0}^{n} (T_{opt}(\Lambda_i) + \tau_{comm}(P)) - \tau_{comm}(P) \\
&\leq c \cdot \sqrt{3} \cdot \sum_{i=0}^{n} \left(T_{opt}(\Lambda_i) + \frac{W(\Lambda_i)}{P} (\frac{1}{\mu(G,P)} - 1) \right) - \tau_{comm}(P) \\
&\leq c \cdot \sqrt{3} \cdot \sum_{i=0}^{n} \frac{T_{opt}(\Lambda_i)}{\mu(G,P)} - \tau_{comm}(P) \text{ since } W(\Lambda_i)/P \leq T_{opt}(\Lambda_i) \\
&\leq c \cdot \sqrt{3} \cdot T_{\mathsf{opt}}(G) - \tau_{comm}(P)
\end{aligned}
$$

CASE $k > 1$: We argue similar as in the case $k = 1$. By induction hypothesis, we have for any $v \in V$ a schedule s_v such that $TIME(s_v) \leq c \cdot 3^{(k-1)/2} \cdot T_{\mathsf{opt}}(s_v)/\mu(G,P) - \tau_{comm}(P)$. Using[4] we obtain $TIME(s_i) \leq c \cdot 3^{k/2} \cdot T_{\mathsf{opt}}(\Lambda_i)/\mu(G,P) - \tau_{comm}(P)$ for the schedule s_i of layer Λ_i. With these observations, we calculate

$$
\begin{aligned}
TIME(s) &\leq \sum_{i=0}^{n} (TIME(s_i) + \tau_{comm}(P)) - \tau_{comm}(P) \text{cf. case } k = 1 \\
&\leq \sum_{i=0}^{n} c \cdot 3^{k/2} \frac{T_{\mathsf{opt}}(\Lambda_i)}{\mu(G,P)} - \tau_{comm}(P) \qquad \text{see above} \\
&= c \cdot 3^{k/2} \cdot \sum_{i=0}^{n} \frac{T_{\mathsf{opt}}(\Lambda_i)}{\mu(G,P)} - \tau_{comm}(P) \\
&\leq c \cdot 3^{k/2} \frac{T_{\mathsf{opt}}(G)}{\mu(G,P)} - \tau_{comm}(P)
\end{aligned}
$$

Remark 1. The bounds can be improved if a better approximation algorithm for scheduling independent malleable tasks is used. If there is an algorithm guaranteeing an approximation factor δ then the factor $3^{k/2}$ can be replaced by δ^k.

4 Conclusions

This paper discussed systems of data-parallel applications requiring high performance. Additionally, applications could be added/removed dynamically. In our scenario, the system architecture could even change due to the results of applications.

We introduced a Dynamic Service Architecture for these systems, based on static compositions and optimizations, but also allows for high performance and flexibility, by use of a lookahead scheduling mechanism. In order to realize the lookahead scheduling, malleable task scheduling is required. The lookahead scheduling and the results in malleable task scheduling are the main contributions of this paper.

Future work include an implementation of the DSA for our test bed [1]. Here, we are are also concerned with practical questions like administrating and prioritizing applications.

On a theoretical level, we are interested in extending our cost model towards the compilation and scheduling processes of the applications. Together with a modeling of the expectations of different system events, we might then be able to prioritize the creation of specific evolved systems including even the creation of systems for more than one evolution step in the future.

References

1. J. Andersson, M. Ericsson, and W. Löwe. An adaptivehigh-performance service architecture. In *Software Composition Workshop (SC) at ETAPS'04*. Electronic Notes in Theoretical Computer Science (ENTCS), 2004.
2. D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a realistic model of parallel computation. In *4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP 93)*, pages 1–12, 1993. published in: SIGPLAN Notices (28) 7.
3. W. Löwe and W. Zimmermann. Scheduling balanced task-graphs to logp-machines. *Parallel Computing*, 26(9):1083–1108, 2000.
4. G. Mounie, C. Rapine, and D. Trystram. Efficient approximation algorithms for scheduling malleable tasks. In *11th ACM Symposium on Parallel Algorithms and Architectures SPAA'99*, pages 23–32. ACM Press, 1999.
5. Wolf Zimmermann and Welf Löwe. An approach to machine-independent parallel programming. In *CONPAR '94, Parallel Processing*, volume 854 of *LNCS*, pages 277–288. Springer, 1994.

[1] *LOFAR (LOw Frequency ARray), www.lofar.org* is a radio infrastructure aiming at multidisciplinary research of astronomers, cosmologists, physicists, climatologists, cosmologists, radio scientists, and IT researchers. It consists of geographically distributed digital sensors connected to computing nodes with a high-speed network. Sensors are distributed over distances of 400 km and the system will produce data at a rate 25 Tbits/s. The Swedish initiative *LOIS (LOFAR Outrigger Scandinavia)* aims, among others, at extending and enhancing the IT infrastructure capabilities of LOFAR.