Scheduling Interactive HPC Applications

Vladimir Nikolov, Stefan Bonfert
Inst. of Inform. Resource Management
Ulm University, Germany
Email: vladimir.nikolov@uni-ulm.de

Eugen Frasch, Franz J. Hauck
Inst. of Distributed Systems
Ulm University, Germany

I. Introduction and General Problem Overview

While modern real-time systems are commonly associated with embedded devices and limited computing power, they recently receive increased attention in the field of high performance computing (HPC) [1]. Many HPC applications perform some form of simulation, e.g. a weather forecast, fluid dynamics or heat dissipation. Typically, a user triggers these simulations with a set of input parameters and collects their output after they finished execution. However, interactive HPC simulations enable users to change parameters while the simulation is running and get immediate feedback. For example, the user might alter the shape of a car during an airflow simulation, or dynamically fine-tune its traction control system for certain simulated ground conditions. To enable interactivity, applications must be highly responsive and should meet tight timing constraints for a continuous user interaction loop, i.e. user input, intermediate processing and result visualization (e.g. with a certain frame rate).

HPC systems are typically realized as distributed clusters of throughput-optimized compute nodes with a high-speed interconnect and a job scheduler. Like in a typical batch system users submit individual jobs with a fixed reservation for *exclusive resources* (e.g. a set of compute nodes) for a certain time period. Modern schedulers like Slurm or Moab perform a simple cluster-wide admission control on job granularity, i.e. assign available resources to individual jobs. However, fixed reservations require a-priori knowledge about applications peculiarity, processing demands and completion time. HPC simulations in turn are compute intensive and highly parallel applications which naturally work in an iterative *fork-join* fashion (see Fig. 1). They periodically (or sporadically) spawn many parallel tasks which are distributed over the reserved resource set (fork). On completion they are synchronized at a common barrier (join) where partial results are gathered and analyzed. This procedure repeats with a dynamic data-preparation phase between a join and the next fork phase.

Technically, this execution model is currently supported by standards like MPI, for inter-process communication and data transmission, and OpenMP, for node-local task parallelism. However, from the perspective of interactive HPC simulations the model is insufficient. Assuming that user input will be incorporated and processed only during each data preparation phase, i.e. after a join and before the next iterative fork, a real-time execution model for the compute intensive parts is required. Thus, a common relative deadline for the parallel computations should be respected on every periodic/sporadic fork. Only then a certain processing rate and the required interactiveness of the simulation can be supported and guaranteed.

Obviously, the current fixed reservation model is also obsolete for interactive applications whose termination is now controlled by the user. Since traditionally resources are reserved exclusively on a per-job basis, compute nodes remain idle between every join and the next fork. A more sophisticated resource management strategy could allow for interleaved job execution on the same compute nodes, which would fill the waiting times and rise the node utilization factor and thus the throughput of the whole cluster. The respective strategy will be discussed in the following.

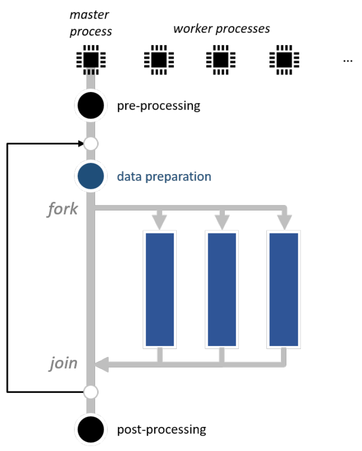

Fig. 1. Fork-Join Execution Model

II. Presupposition

The need for a general theory covering both high performance computing and real-time theory has been answered with the *real-time divisible load theory* (RT-DLT) [2]. It relies on a system model that closely resembles a typical HPC cluster setup, where one head node receives jobs from users, performs scheduling and distributes the workload on available compute nodes. For workload partitioning and distribution, RT-DLT relies on the classical *divisible load theory* (DLT) [3], that assumes a relation between the amount of data processed by a task and its execution time. Given a divisible job RT-DLT solves the scheduling problems of (a) how to divide the job among a set of processors in order to minimize its completion time, and (b) what is the minimum number of processors assigned to that job in order to meet its deadline. In the literature RT-DLT has been further elaborated for heterogeneous clusters and different processor ready times [4]. However, RT-DLT does not respect the typical iterative fork-join behaviour of HPC-applications. It considers only aperiodic workloads which are divided once, distributed to an exclusive set of resources and run to completion. In case of periodic job submissions conventional RT-DLT would lead to unnecessary re-scheduling, communication and task initialization phases on variable compute nodes.

III. Hierarchical Job Scheduling

We propose a hierarchical scheduling approach on top of RT-DLT for *periodic divisible jobs* $J_i = (A_i, T_i, \sigma_i, D_i)$, with a fixed inter-arrival time T_i (period), a maximal total data size σ_i per iteration and a relative deadline D_i. The basic idea is to sync the occurrence of periodic application forks with the activation sequence of its distributed tasks. However, assuming varying fork conditions as well as data preparation and transmission times, a sporadic model can be established as well. Figure 2 resembles the general scheduling architecture.

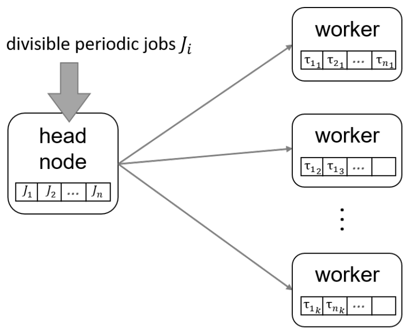
divisible periodic jobs J_i
head node
J_1 J_2 ... J_n
worker
τ_{1_1} τ_{2_1} ... τ_{n_1}
worker
τ_{1_2} τ_{1_3} ...
worker
τ_{1_k} τ_{n_k} ...

Fig. 2. Hierarchical Job Scheduling

Similar to RT-DLT a head node accepts and schedules jobs, but now several jobs may be hosted and executed at the same time on the same set of resources (*interleaved job execution*). When a job arrives and is accepted applying classical RT-DLT now relates to a particular job iteration. Following the rules in [3], [4] this results in the required number k_i of worker nodes for each job iteration in order to meet D_i and the minimal processing time C_i for the whole iteration (i.e., also for each node). Here for simplicity two assumptions were made:

- All nodes have homogeneous hardware; however, heterogeneous solutions for RT-DLT could be incorporated as well.
- The amount of processed data for each job iteration remains $\leq \sigma_i$, thus, C_i is a maximum estimation of iterations' processing costs; otherwise, we would have to reschedule the Job (i.e. recompute the RT-DLT) at each iteration.

Consequently, J_i then decomposes to a set of k_i periodic tasks of the form $\tau_{i,k} = (A_i, T_i, C_i, D_i)$ with equal periods T_i (equal to their job-period), costs C_i and relative deadlines D_i. These tasks can be now deployed to various worker nodes having local scheduling queues (cf. Fig. 2). Consequently, a node can host and execute several tasks of different or even the same job in parallel.

Let's assume the workers implement an EDF discipline, which allows an exact feasibility analysis for a node utilization factor up to 1 (i.e. 100%). As a result, the overall utilization U and the available slack time S can be computed for each particular node and its local task configuration. In case of $D_i \leq T_i$ a processor-demand–based feasibility test may even be used, e.g. approximation of tasks demand bound function [5]. Thereby, a synchronous start of the local task set can be assumed in oder to neglect actual activation phases of the tasks. However, meeting of $\tau_{i,k}$'s own deadlines, and thus the relative job deadline, is guaranteed on worker-node level as long as local feasibility can be proved. The node-specific utilization (or processor demand) and slack time are reported to the head node and used for further placement decisions within the cluster. These values have to be updated only if the local task configuration changes or their cost approximations are violated.

Let's assume, C_i are cost upper bounds for the periodic tasks and each node monitors its internal tasks' actual execution times; here a dynamic cost approximation model can be used like the one presented in [6]. If a task $\tau_{i,k}$ exceeds its current estimate C_i the actual schedule of the respective job J_i might be corrupted and meeting further iteration deadlines not guaranteed. In this case, the head node is required to compute a new schedule and redistribute the tasks if necessary.

IV. Discussion and Open Problems

It is obvious that with our hierarchical job scheduling model potential idle times of the workers and RT-DLT based rescheduling costs (i.e. job division, node initialization, communication) can be minimized. This model is particularly useful for interactive fork-join based parallel applications, which traditionally would occupy nodes idling during sequential application phases, data preparation and transmission (until the next fork). Even short blocking effects of tasks on the nodes can be masked locally with sensitive work of other jobs and tasks. Thus, the overall cluster utilization is increased while still being able to respect relative job deadlines, even for each fork-join phase.

Nevertheless, some problems still remain open for discussion. For node-local and cluster overload management we aim to extend our model with a dynamic quality control on task-level, like the one proposed in [6]. Here, quality decisions are opposed by potential task migrations and must apply in a distributed fashion. Apart from this some technical challenges exist like (a) how to sync data delivery to workers with the periods of their internal tasks, (b) how to capture worker local application state during task migrations, and (c) how to realize an optimal task placement, e.g. for minimal comm. costs on join phases.

References

[1] I. Alvarado, "Real-Time High-Performance Computing with NI LabVIEW," http://www.ni.com/white-paper/11972/en, 2012, [Online].
[2] X. Lin, Y. Lu, J. Deogun, and S. Goddard, "Real-time divisible load scheduling for cluster computing," in *RTAS'07*, April 2007, pp. 303–314.
[3] V. Bharadwaj, D. Ghose, and T. G. Robertazzi, "Divisible load theory: A new paradigm for load scheduling in distributed systems," *Cluster Computing*, vol. 6, no. 1, pp. 7–17, Jan. 2003.
[4] S. Chuprat and S. Baruah, "Scheduling divisible real-time loads on clusters with varying processor start times," in *Proceedings of the 2008 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, ser. RTCSA '08. IEEE Computer Society, 2008, pp. 15–24.
[5] S. Baruah, A. Mok, and L. Rosier, "Preemptively scheduling hard-real-time sporadic tasks on one processor," in *RTSS*, 1990.
[6] V. Nikolov, S. Wesner, E. Frasch, and F. Hauck, "A hierarchical scheduling model for dynamic soft-realtime systems," in *Proc. of the 29th Euromicro Conference on Real-Time Systems (ECRTS)*, 2017.