FAUST

A Framework for Adaptive Ubiquitous Scalable Tasks

Abstract

Contents

1 Random Thoughts

1.1 Modeling Job Dependencies

The overall goal of the FAUST framework is to schedule a given set of jobs on a number of distributed resources as effective as possible. Effectiveness in our case means minimum *makespan*[1] scheduling or time to completion. In case of an application which consists of a set of independent jobs (embarrassingly parallel EP), scheduling is rather trivial: execute as many jobs as possible at the same time on all available resources. An example for such an application would be a parameter sweep which generates and executes a set of independent model instances with different input parameters.

However, lots of distributed applications do not fall into the category of EP applications. Jobs often require communication with other jobs or they may rely on data that has to be generated by other jobs. Message-passing (e.g. MPI) as well as distributed workflows are good examples for these type of applications. Unfortunately, scheduling becomes way more complex in this case, since it has to take not only the availability of resources but also things like interconnect bandwidth, shared filesystems, etc. into account to minimize the overhead exposed by dependencies.

In this section, we try to identify different types of job dependencies, describe how to model them on application level and discuss the implications for possible minimum makespan scheduling algorithms.

1.1.1 Types of Dependencies

So far, we identified two types of dependencies in distributed applications that are relevant for job scheduling and placement. These are communication dependencies and data dependencies.

Data Dependencies occur whenever a job requires data that is generated by another job or a set of jobs. Imagine an image processing application (figure 1) that splits up an image into several regions and applies a filter to each of the regions in parallel. Another job takes the processed fragments and puts them back together. This job depends on the output generated by the filter instances.

Communication Dependencies

[1] The makespan of a schedule is its total execution time.

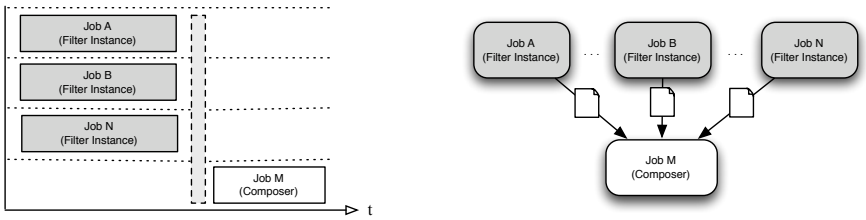

Figure 1: Example of a dependency graph *(r)* for an image processing application where job *M* depends on the data generated by jobs *A...N*. The grey vertical bar in the scheduling scheme *(l)* represents the time overhead generated by data transfer.

1.1.2 Describing Dependencies

A proper description of job dependencies and their attributes on application level is the key for effective scheduling and placement of jobs.

Data Dependencies expose a potential data transfer overhead. A scheduling algorithm has to decide wether it either should move the data to the computation or the computation to the data (place the dependent job as *'close'* to the data as possible). To be able to make this decision, the following application attributes have to be specified:

- Expected runtime of
- Amount of data generated by a

The FAUST framework provides an interface to describe data dependencies in applications through the *job submission* interface. In case of the example image processing application described above, this could look like like to following code fragment:

Describing Data Dependencies

```
job::description filter_jd;
filter_jd.set_attribute(walltime, "10.0");
filter_jd.set_attribute(data_volume, "0.5GB");

std::vector<std::string> filter_desc;
for(int i=1; i<10; ++i)
  filter_desc.push_back(filter_jd); // create 10 filter instances

job::service s;
```

```
job::group filters = s.create_job_group(filter_desc);

// create the composer job which has a DATA dependency with the
// filter job group.
job::description composer_jd;
job::job composer = s.create_job(composer_jd, filters, type::DATA);

s.schedule();
```

2 Appendix

3 References

References