January 27, 2009 Ole Weidner (oweidner@cct.lsu.edu)

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 types 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 job 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. We distinguish between dependencies that rely on data availability and dependencies that rely on communication:

Data Dependencies occur whenever a job requires data that is generated by another job or set of jobs. Imagine for example an image processing application (Figure **??**) 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 occur whenever two or more jobs need to exchange information while they are running. Imagine a 2D heat-tansfer ap-

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

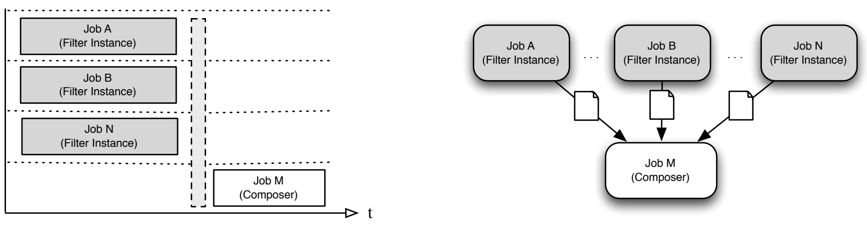

Figure 1: Example of a dependency graph *(r)* for an image processing application where job M depends on the data generated by jobs $A ... N$. The grey vertical bar in the scheduling scheme *(l)* represents the time overhead generated by data transfer.

plication (Figure **??**) that splits up the problem space in 4 regions and maps them to 4 different jobs (domain decomposition). Communication has to occur whenever the heat transfers across domain boundaries. This concept is also known as ghost-zone exchange and a very well known concept in MPI.

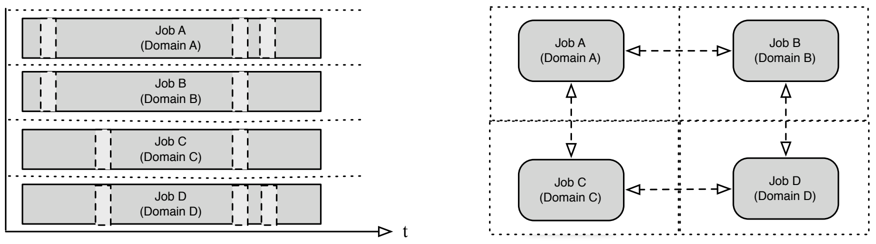

Figure 2: Example of a dependency graph *(r)* for set of communicating jobs in a domain decomposition application. The grey vertical bars within the jobs in the scheduling scheme *(l)* represent the time overhead generated by communication.

1.1.2 Describing Dependencies

Describing just the dependencies and type of dependencies (data or communication) between jobs enables a scheduler to execute the jobs in a way that satisfies the dependencies. However, without additional information about the jobs and the logical dependencies, a scheduling algorithm won't be able to *place* the jobs efficiently on the distributed resources. These attributes usually can't be extracted from the application automatically. They have to be described explicitly on application level. We identify a minimum set of these attributes and show how they can be described using the FAUST API.

Data Dependencies expose a potential data transfer overhead. A scheduling algorithm has to decide wether it should either move the data to the computation or the computation to the data (place the dependent job as close[2] to the data as possible). To be able to make this decision, the following information has to be provided on application level:

- **Expected runtime** of the jobs that are part of the dependency.

  `faust::attribute::walltime`

- **Expected amount of data** generated by a job.

  `faust::attribute::data_volume`

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 the following (simplified) 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();

```

Communication Dependencies expose a potential communication overhead. A scheduling algorithm has to decide wether it should place the jobs on resource which are connected via high-bandwith interconnects or if processing power for the individual jobs is more important than communication. The following application attributes can be specified on application level to help the scheduler to make the right decisions:

[2] *Close* in this context is defined as the interconnect bandwidth between two locations.

- **Computation/communication ratio**

  `faust::attribute::comm_comp_ratio`

- **Communication pattern** (regular, irregular, ...)

  `faust::attribute::comm_pattern`

FIXME: I'm still struggling if these are the right attributes and if it's feasible to describe them on application level. Reading some papers about this might help...

2 Implementation

2.1 Overview

2.2 Interface

2.3 Scheduling Engine

2.4 Agents

The Agents are used to provide the Scheduling Engine as well as the programmer (through the `faust::resmon` API) with periodic informations about all participating systems that are relevant for scheduling descissions and job execution. *FAUST* Agents are realized as independent command line applications that run on the execution hosts (usually on the head or gateway nodes) and report vital system and status informations like queue status, network load, filesystem availability, etc. back to a *FAUST* application instance.

Figure 3: Example of a *FAUST* application instance running agents on three execution hosts. To ensure application persitency, the communication between the application and the agents flows through a proxy database.

Besides reporting system informations to the application, agents can optionally act as job submission endpoints. In this scenario, an application sends jobs directly to the agents for execution and not to the system's job service (like Globus, GridSAM, PBS, etc). This will usually happen, if the application scheduler (or the programmer) decides to apply 'hijacking' strategies like *Glide-In* which require circumvention of local queueing systems.

2.4.1 Database

A critical design descission was that all communication between a *FAUST* application and its agents must be routed through a proxy database to ensure global persitency in a distributed environment. This indirect communication between agents and application was motivated by two important requirements during the design process:

- A *FAUST* application must have the ability to disconnect and reconnect from its infrastructure whithout the need to abort and restart.

- A *FAUST* application might run in a restricted namespace (eg. firewalled). A communication proxy can help to avoid this restriction.
- The data collected by the agents might be of interest to other application or services. A central database allows other clients (Web-Services, etc.) to harvest and use this data.

The agents use the SAGA *Advert Service* and its *PostgreSQL*-based middleware adaptor to read and write hierarchical entries from or to a centralized datbase. The structure of the database (as shown in Figure **??**) is known to the agents and the application framework. In case the applications wants to read system informations from a certain agent, it simply reads the entries to which the agent periodically writes its information. If the application framework wants to send a command to an agent, it writes a command to the entry in which the agent periodically looks for new commands.

Although the communication between agents and application is not very dense, this approach introduces a certain amount communication overhead that can slow down an application - especially with the current implementation of the *PostgreSQL*-based SAGA middleware adaptor. A simple improvement would be the development of a high-performance SAGA Advert adaptor which would allow for higher data throughput without the need to change the *FAUST* framework. Benchmarks for the current *PostgreSQL*-based implementation can be found in section **??**.

2.4.2 Mode of Operation

The *FAUST* Agents are transparently deployed and started by a *FAUST* application as soon as a new `faust::service` object gets instantiated. Each `faust::service` instance is defined by a set of resources which represent possible target hosts for job submission. Each `faust::resource` has its own Agent for management, monitoring and information retreival.

Unfortunately, different distributed infrastructures require different techniques to extract the informations that are required for effective scheduling. Although there are several more or less well defined interfaces to query these informations like *NWS* or *BQP*, it appears that in practice their existence and proper operation cannot be assumed. For this reason, agents require a relatively rich set of initial system informations that are used to generate subsequent runtime informations and performance predictions. Once deployed and started, an Agent follows the (simplified) execution scheme shown in Figure **??** .

The agents provide basic logging and fault tolerance mechanisms which are crucial in a capricious distributed environment.

```
BEGIN
 IF provided host description is usable on this machine THEN
   REPEAT
      Gather informations and write to database
     IF Resource reservation request THEN
        Try to reserve resources through queueing system
     ENDIF
     IF Job execution request THEN
       IF Resources available THEN
         Execute job
       ELSE
          Report that no resources are available
       ENDIF
     ENDIF
   UNTIL  Termination request received
 ELSE
    Report error and terminate
 ENDIF
END
```

Figure 4: Pseudocode describing the Agent's mode of operation.

2.4.3 Host Files and Agent Repositories

3 Appendix

4 References