University of Alberta

Library Release Form

Name of Author: Mark Goldenberg

Title of Thesis: TrellisDAG: A System for Structured DAG Scheduling

Degree: Master of Science

Year this Degree Granted: 2003

Mark Goldenberg
Ap. 402
10101 Saskatchewan Drive
Edmonton, Alberta, T6E 4R6
Canada

Date: _____

University of Alberta

TRELLISDAG: A SYSTEM FOR STRUCTURED DAG SCHEDULING

by

Mark Goldenberg

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Master of Science**.

Department of Computing Science

Edmonton, Alberta
Spring 2003

University of Alberta

Faculty of Graduate Studies and Research

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled **TrellisDAG: A System for Structured DAG Scheduling** submitted by Mark Goldenberg in partial fulfillment of the requirements for the degree of **Master of Science**.

Paul Lu
Co-Supervisor

Jonathan Schaeffer
Co-Supervisor

Peter Minev

Martin Müller

Date: _____

Abstract

The progress of research in many areas of science, including physics, chemistry, engineering and others, depends on the ability of the researchers to perform massive computations on the order of month and years of CPU time.

We have designed and implemented TrellisDAG – a system that combines the use of placeholder scheduling and a subsystem for describing workflows to provide novel mechanisms for computing non-trivial workloads with inter-job dependencies. This combination results in TrellisDAG enjoying the following properties:

1. Ability to use resources across multiple administrative domains,
2. Automatic load balancing across all of the used resources,
3. High utilization of the opportunities for concurrent execution while respecting inter-job dependencies, and
4. Convenience of monitoring of the computation.

The design of TrellisDAG allows it to handle many prospective applications and we used computing the checkers end-game databases as a motivating application and proof-of-concept.

Acknowledgements

I would like to thank my co-supervisors, Paul and Jonathan, for guidance, advice, and invaluable moral support that they have given me throughout the program.

Thank you to Chris Pinchak and all fellow graduate students in the Software Systems research lab. Their support and friendship were of great importance.

I am forever grateful to my family in Edmonton – Isaak, Lora, Luba and Ilia Agranovitch – for everything they did for me.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The progress of research in many areas of science, including physics, chemistry, engineering and others, depends on the ability of the researchers to perform massive computations on the order of month and years of CPU time. Examples of the problems that require computation of this scale include Earthquake Ground Motion Modeling and solving the Non-linear Einstein equations describing General Relativity. These and other challenging computational problems are described at the HPCC Grand Challenges web site [15]. Often, such computations consist of the execution of many programs or of many executions of one program with different parameters, or a combination of the two. We refer to an executable unit of computation as a *job*.

The important observation here is that there is an opportunity to speed up the computation by running jobs on different (and possibly remote) processors. Suppose that we have jobs A, B and C that can be started simultaneously without affecting the overall correctness of the computation. If we have only one processor, then executing all three jobs at once (i.e. multiprogramming) may not make much sense unless there is a specific mix of computation and I/O. However, if we have three processors, then it makes sense to execute the jobs concurrently (or, as we also say, in parallel).

As simple as it may seem, there are several tasks involved in such a decision:

1. Identifying that the jobs A, B and C can be run in parallel,

2. Recognizing that there are three processors at the moment that are ready to

execute the jobs, and

3. Mapping the jobs to the processors – the task that we refer to as *scheduling*.

The Trellis project is concerned with performing the above tasks automatically, harnessing all of the available processors to speed up the user's computation. The CISS-1 and CISS-2 [33] experiments showed how the recently introduced technique of placeholder scheduling [30, 31, 32] automates the tasks described above in the cases when all of the jobs can be executed in parallel. In this thesis, we consider the same tasks for the more general case when some jobs can be executed only after other jobs are completed, i.e. when there are inter-job or, as we also say, *workflow* dependencies between the jobs of the computation.

1.1 Contributions of this thesis

To properly put the contributions of this thesis in context, it is important to understand two standard notions from the area of operating systems: *mechanisms* and *policies*. Mechanisms provide the infrastructure or the means for carrying out a task, but they do not specify how exactly the task is to be carried out. Policies use mechanisms to carry out the task. Suppose that our task is to keep the users of a large system from using too much disk storage. The file system provides mechanisms for achieving that goal: the administrator's ability to define disk quotas. However, the exact quotas are a matter of policy. The policies can change (sometimes quite often), but the mechanisms do not need to change in order to put a new policy in place (e.g. the file system does not need to be rewritten in order to change the users' disk quotas).

All of the contributions of this thesis can be classified as mechanisms. Developing policies that take advantage of the mechanisms introduced in this thesis is an interesting direction for future work.

The contributions of this thesis can be summarized as follows:

1. Design and implementation of TrellisDAG – a system that can use the technique of placeholder scheduling for the case of inter-job dependencies,

2. Novel mechanisms for representing the workflow in terms that are natural for the application,

3. Novel mechanisms for job scheduling in the context of placeholder scheduling with inter-job dependencies, and

4. An evaluation of the effectiveness of our approach using a non-trivial application.

The technique of placeholder scheduling allows one to build an *overlay metacomputer* on top of the existing infrastructure [32]. Suppose for example, that a researcher has access to processors at the University of Alberta and the University of British Columbia (see Figure 1.1). Each of these universities define their own independent (of each other) policies for using resources and, because of that, we say that the two universities represent two different *administrative domains*. The fact that there are multiple administrative domains makes it difficult for a computational scientist to use the available processors as a single computational resource. Therefore, we need an infrastructure on top of the existing one, where the resources in the two different institutions are abstracted. This infrastructure is called an overlay metacomputer [32]. This metacomputer consists of the processors at the two universities, but is logically represented to the user as one point of control. As well, placeholder scheduling provides a natural mechanism for load balancing of jobs between the individual resources constituting the metacomputer.

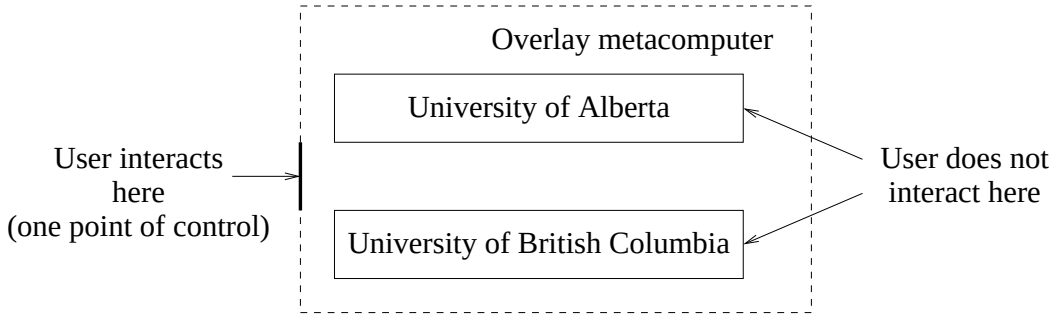

Figure 1.1: Two administrative domains and an overlay metacomputer.

TrellisDAG utilizes the technique of placeholder scheduling and therefore enjoys both the abstraction of resources provided by the overlay metacomputer and

the load balancing properties mentioned above.

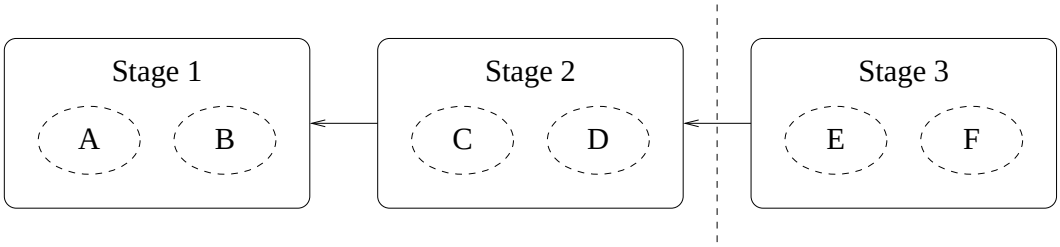

Figure 1.2: A 3-stage computation. It is convenient to inquire the status of each stage. If the second stage resulted in errors, we may want to disable the third stage and rerun starting from the second stage.

TrellisDAG enhances the convenience of monitoring and administering the computation by providing the user with a tool to translate the set of inter-dependent jobs into a hierarchical structure with the naming conventions that are natural for the application domain. Suppose for example, that the computation is a 3-stage simulation, where the first stage is represented by jobs A and B, the second stage is represented by jobs C and D, and the third stage is represented by jobs E and F as shown in Figure 1.2. The following are examples of natural tasks of monitoring and administering of such a computation:

1. Querying the status of the first stage of the computation, e.g. is it completed?

2. Making the system execute only the first two stages of the computation, in effect disabling the third stage.

3. Redoing the computation starting from the second stage.

TrellisDAG makes such services possible by providing a mechanism for a high-level description of the workflow, in which the user can define named groups of jobs and specify collective dependencies between the defined groups.

Finally, TrellisDAG provides mechanisms for scheduling:

1. The user can submit all of the jobs and dependencies at once. The importance of submitting the workflow dependencies can be demonstrated by an example. Suppose that the computation consists of jobs A, B and C and that the

jobs B and C can only start after A is done (as shown schematically in Figure 1.3). It is possible to envision a system where the user is only required to submit the jobs in some legal order, for example A, B, C in our case. However, such a system would have no way of figuring out that the jobs B and C can be executed in parallel and would not take advantage of this opportunity. Submitting **all** jobs and dependencies at once gives the scheduler more information and makes it possible to look ahead before making scheduling decisions.

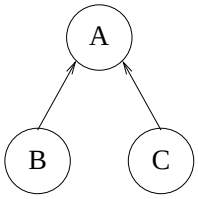

Figure 1.3: Submitting the dependencies helps, as in this case.

2. TrellisDAG provides a single point for describing the scheduling policies. The placeholder scheduling maps the ready jobs to resources. However, it does not restrain our freedom of choosing the jobs to be run. There is a procedure that gets invoked to choose the next job to run. All of the information stored about jobs is available to that procedure. Also, the procedure knows where the chosen job will be executed.

3. TrellisDAG provides a flexible mechanism by which attributes may be associated with individual jobs. The more information the scheduler has, the better scheduling decisions it may make. Of special importance is the knowledge of the length of the jobs. For example, suppose we have processors P_1 and P_2, and jobs A, B and C, with no inter-job dependencies, that run for 10, 20, and 40 seconds, respectively. If the scheduler does not know *a priori* the running times of the jobs, it may schedule A and B to run on P_1 and P_2 respectively, and, after 10 seconds, schedule C to run on P_1, resulting in the total time of 50 seconds. However, if the user can estimate the time and submit these times together with the jobs, then the scheduler can easily come up with a schedule

that takes only 40 seconds by choosing C to run first. Although such an attribute is not used for scheduling in this version of the system, the mechanism is in place.

4. Stores the history information associated with the workflow. The scheduler may use learning in order to improve the overall computation time from one trial to another. Our system provides mechanism for this capability by storing the relevant history information about the computation, such as the start times of jobs, the completion times of jobs, and the resources that were used for computing the individual jobs. Developing scheduling policies that use this mechanism is an interesting direction for future work.

We demonstrate the effectiveness of our approach by using TrellisDAG for a non-trivial application – computing the checkers end-game databases (see Chapter 3) – and show that:

1. Specifying the jobs and inter-job dependencies using our system is a straightforward task,

2. Using the system results in good utilization of resources and the application-specific opportunities for inter-job concurrency, and

3. The overhead introduced by using TrellisDAG is small and does not significantly affect the overall computation time for the applications of interest.

1.2 Overview

Chapter 2 surveys the relevant work and puts the work of this thesis in perspective.

In Chapter 3, we describe the application that originally provided motivation for this research and was later used as a proof-of-concept. We use the same application as a running example in Chapters 4 and 5.

Chapter 4 is a user guide to the system. It provides the reader with all the necessary information to understand the system at a conceptual level and be able to use it.

Chapter 5 goes into the implementation details of the system.

In Chapter 6, we describe our experiments and demonstrate that the system achieves the design goals outlined above and exhibits other good properties.

Finally, in Chapter 7, we conclude and list some directions for future work.

Chapter 2

Background and Relevant Work

High-performance computing (HPC) is the area of computing science that is concerned with achieving maximal computational benefits by exploiting the opportunities for concurrent execution. We can distinguish two types of concurrency:

1. Intra-job concurrency refers to executing two or more different instructions of one job simultaneously on different processors, e.g. Single Program Multiple Data (SPMD) parallel computing.

2. Inter-job concurrency refers to executing two or more distinct jobs on different processors.

This thesis is concerned with the applications that exhibit opportunities for inter-job concurrency and only considers sequential jobs. Computational science provides many examples of such applications (e.g. the computational chemistry application used in the Canadian Internetworked Scientific Supercomputer (CISS) experiment [33] and computing checkers end-game databases [19]). Most commonly, these applications exhibit the following properties that we will take advantage of:

1. Non-interactivity (batch jobs). A job is usually started with a number of command-line parameters and/or with a file given to it as an input. These parameters and/or file completely specify what computation should be performed by the job and no interaction with the user is required. This property is important when we want to execute the job remotely and thus detach it from the terminal through which it could possibly interact with the user.

2. Long run times. The jobs that constitute the applications of computational science tend to be long – on the order of hours or days of CPU time per job. Because of the coarse granularity, using mechanisms for scheduling that have a potential of incurring considerable overhead is not a concern. At the same time, scheduling policies based on such mechanisms may considerably speed up the computation.

3. Large number of jobs. It is common that a scientific computation consists of running one or several executables over and over again with different sets of parameters or inputs. For example, CISS-1 [33] computed 7,592 jobs during a 24-hour period; the checkers computation [19] that we demonstrate in Section 6.8 involves 1,276 jobs. In the presence of a large number of jobs, good scheduling decisions are especially important, because they can have a significant cumulative effect on the overall time taken by the computation.

4. Inter-job dependencies. Sometimes, not all of the jobs of the application can run concurrently. That is, there can be ordering constraints called *dependencies*. Usually such constraints exist because the result of one job is used by another job of the computation. The workflow in Figure 1.3 is an example of a small workflow having dependencies. One of the applications exhibiting this property inspired the work on this thesis (see Chapter 3).

This survey consists of two major parts that could roughly be separated as referring to either mechanisms or policies.

1. Mechanisms. Sections 2.1-2.4 describe systems that provide mechanisms for automatically executing the jobs of the computation on different and, possibly, remote processors. Some of these systems provide a number of mechanisms similar to those described as the contributions of this thesis. The major difference is that our system uses placeholder scheduling as its underlying infrastructure.

2. Policies. The order in which jobs are executed and the mapping of jobs to resources can have a large impact on the overall computation time. This issue

becomes especially interesting when the computation has inter-job dependencies. Section 2.5 describes the work that has been done in the area of scheduling and, in particular, *DAG Scheduling* [17], where DAG refers to the directed acyclic graph used to represent the workflow with its dependencies.

2.1 Batch schedulers

The first widely used batch scheduler was the Network Queuing System (NQS) [2] developed by the NASA Ames Research Center. The main purpose of the system was to give the users of slower machines the opportunity to submit their jobs to a faster machine. There were multiple users competing for CPU cycles of the fast machine and the system helped by providing several services:

1. The job queue. The submitted jobs were put in a queue similar to a printer queue, thus making sure that the jobs of different users were executed in turn and did not interfere with each other, and

2. Restriction policies. With every queue there were associated quotas that limit the amount of resources that the individual jobs could use up. For example, the administrator could specify that a job should not run for more than 2 hours, or that a program should not use more than 16 MB of memory, etc.

Although NQS was usually used in the environment with only one or a few high-performance machines on which most of the computation was done, the ultimate purpose of the system was to harness the available computational resources and make these resources available to the users.

A descendant of NQS, the Portable Batch System (PBS) [34], is widely used today. Relating to our work, PBS provides the following interesting features:

1. Load balancing. It is possible to install PBS on a cluster of workstations so that there is a central submission point. Then, PBS makes sure to put the jobs on the less loaded machines. If there are no dependencies between the jobs and all jobs are submitted at the same time (e.g. by a submission script), then PBS will do a good job of maximizing the throughput.

2. Job dependencies. Every job is assigned an identity (ID) upon submission. Thus, if the job C depends on jobs A and B that were submitted previously and assigned IDs 1234 and 1235 respectively, then the command for submitting C may look as follows: `qsub -Wdepend=afterok:2534,1235 C.sh`. As seen in this example, PBS also supports basic fault tolerance; namely the `afterok` key word means that the job C will only be executed upon successful completion of its antecedents.

PBS has several limitations that are addressed by our system:

1. PBS is designed to work within one administrative domain, such as a cluster of workstations. Thus, the amount of computational power harnessed by PBS is quite limited. In contrast, TrellisDAG can work with widely dispersed computational resources.

2. Because the jobs are assigned IDs at submission time and knowing them is required to specify dependencies, writing a submission script that takes job dependencies into account is not a trivial task. In contrast, TrellisDAG encourages that the jobs and the dependencies are submitted all at once. Also, the system provides a single location where both the jobs and the dependencies are specified.

Other widely used batch scheduling systems are the Sun Grid Engine (SGE) [38], the Load Sharing Facility [24, 43], and IBM's LoadLeveler [25]. In terms of load balancing and inter-job dependencies, they have similar capabilities with PBS.

As we will explain later in this chapter, placeholder scheduling does not replace the local batch schedulers; instead, it allows our system to take advantage of the good properties of the local batch schedulers and reuse their load balancing functionality.

2.2 Specialized systems

There are systems made to address only a subset of the needs of scientific high-throughput computing. One such system is called Nimrod[1] and is designed to help scientists explore large parametric spaces [1].

Nimrod is a specialized system for two reasons:

1. Job dependencies are of no concern for Nimrod because the parameter sets can be explored in parallel. A workload in which all jobs can be run in parallel without violating any dependencies is called *embarrassingly parallel*.

2. Nimrod can only be applied to problems that are formulated as parameter space exploration.

Nimrod has a feature that is similar to TrellisDAG: there is a single place where the whole computation is defined. The user defines the parameter space to be explored in a special programming-like language. He also defines the operations to be performed before and after exploring each particular parameter set. These operations usually include data transfer.

Nimrod works within one administrative domain. This limitation is addressed by the grid-enabled (see Section 2.4) version of this system called Nimrod/G [41].

2.3 Metacomputing systems

The purpose of a metacomputing system is to represent a set of heterogeneous and, possibly, dispersed computing resources as a unit. Such a system is expected to have the following characteristics:

1. Transparent (for the user) mapping of jobs that were submitted to the metacomputing system to the physical machines,

2. Load balancing. The system has to, in effect, maximize throughput, and

[1]The name Nimrod comes from the Bible. Nimrod, son of Cush, son of Ham, was a great-grandson of Noah who was saved during the Flood. The Bible describes him as "the first to be a mighty man on earth."

3. Monitoring of the jobs being executed.

We note that the batch schedulers described in Section 2.1 exhibit some or all of the properties listed above. However, we typically expect that a metacomputing system can work across multiple administrative domains. The batch schedulers described in the last section can also be installed in a way that covers several administrative domains – at a cost of potentially compromised security. In particular, the clients of a batch scheduler are trusted and, when such a client connects to the scheduler's server, no machine and/or user authentication takes place. This is not the case in placeholder scheduling (see Section 2.3.2), where placeholders connect to the server through an authenticated `ssh` connection.

We will now describe some existing metacomputing systems.

2.3.1 Condor

The Condor[2] High Throughput Computing environment [3, 23] is one of the first metacomputing systems. It was developed at the University of Wisconsin-Madison in the late 1980s to facilitate scientific research that consumed years of CPU time. In 1997, the computing pool of Condor consisted of 300 UNIX workstations delivering on average more than 180 CPU days per day.

Although technically Condor is a batch scheduler, it has several advanced features (described below); moreover, there is an extension of Condor called Condor-G [4] that works across administrative domains.

The system is built on three concepts:

1. **ClassAd.** ClassAds are descriptions of resources and computational tasks. The resources' ClassAds are like advertisements of offers and the computational tasks' ClassAds are like advertisements of demands. Once in a while the ClassAds from two sides are matched and the most suited resources and tasks are paired up.

2. **Remote System Calls.** This is a mechanism by which Condor deals with

[2]Condor is a large vulture; in fact, it is the largest bird in the Western Hemisphere and its wingspan is about 3 meters.

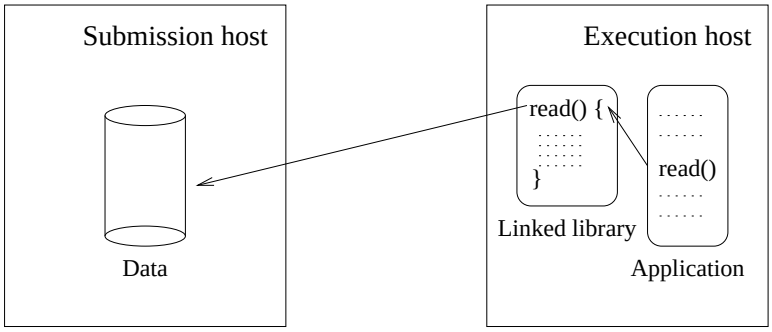

Figure 2.1: Remote system calls in Condor.

the data. The data is always stored on the submission host and the remotely running application needs a means to access that data. In order to do that, the application has to be linked with a special library, which overrides the well-known I/O functions such as `open()` and `read()` to access remote data as shown in Figure 2.1. The interesting consequence of this is that the user does not have to have an account on all the execution hosts and his programs are run under a special account with very limited permissions. The downside is that the submission host may end up overloaded, especially if the application is I/O-bound.

3. **Checkpointing.** A Condor process can save its complete state, periodically and/or on demand. This means that Condor processes can be preempted (i.e. the resources taken up by these processes are reclaimed) and migrated, which is beneficial from the scheduling point of view and provides a mechanism for fault tolerance.

Condor comes with a tool called the *Directed Acyclic Graph Manager* (*DAGMan*) [5]. The tool addresses the problem of job dependencies and allows the user to do something very similar to what our system does: one can represent a hierarchical system of jobs and dependencies using DAGMan scripts. Let us look at a sample DAGMan script:

```
Job A A.condor
Job B B.condor
```

```
Job C C.condor
PARENT A B CHILD C
```

Here, the job C depends on the results of jobs A and B. The names with the `.condor` extension are the scripts representing the jobs. What makes the model interesting is that those scripts can be DAGMan scripts themselves. This is useful because when a group of n_1 jobs can only be started after another group of n_2 jobs has executed, then there are on the order of $n_1 n_2$ dependencies that need to be specified. The model allows us, in effect, to reduce the number of dependencies to one. In our system, we provide a similar functionality through a mechanism by which jobs can be organized in groups, with the following additional advantages:

1. All of the dependencies are specified within one file,
2. The jobs get names automatically, which allows for a large number of jobs, and
3. The information about the groups of jobs is used for monitoring and administrative purposes.

2.3.2 Trellis

The Trellis project [42] aims to develop simple and effective techniques for forming a virtual supercomputer out of a geographically-distributed collection of computing resources. Investigators of the Trellis project identified and developed solutions for two main problems:

1. Distribution of computation. The technique of *placeholder scheduling* [30, 31, 32] provides a mechanism for automatic load balancing of computational jobs across workstations that can be in different administrative domains.
2. Data access. The *Trellis File System* [36] introduces a namespace similar to that used by the UNIX `scp` utility and makes this namespace available to programs through wrappers to the standard I/O functions, such as `open()`

and `read()`. The Trellis File System also supports caching, sparse access, and prefetching.

The technique of placeholder scheduling is very relevant to this thesis, since TrellisDAG is built on top of the technique. Therefore, we provide a more detailed explanation of the technique here. In contrast, we do not use the Trellis File System in this thesis and leave a thorough treatment of data movement by TrellisDAG as a topic for future work.

At a high level, placeholder scheduling is illustrated in Figure 2.2. There is a central host that we will call the *server*. All the jobs (a job is anything that can be executed) are stored on the server (the format of storage is not specified; it can be a plain file, a database, or any other format). There is also a set of programs called *services* that form a layer called the *command-line server*. The command-line server provides means for accessing and modifying the stored jobs in ways that will be described later.

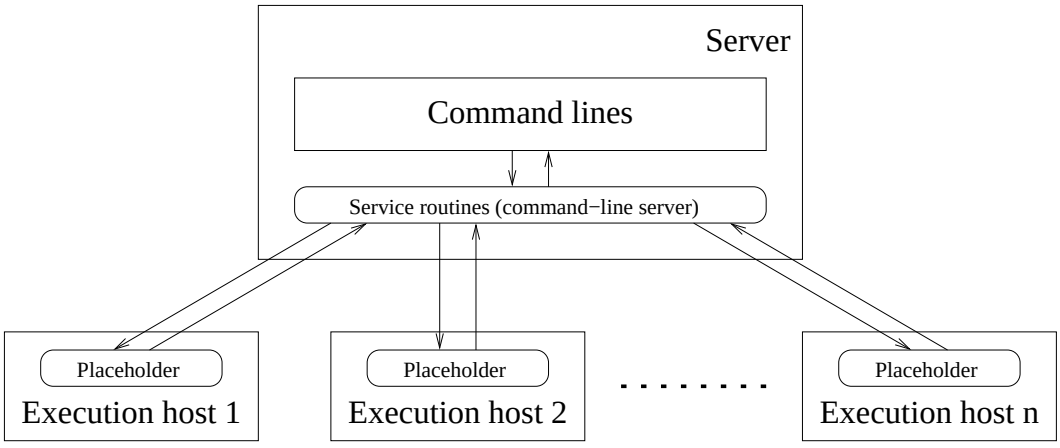

Figure 2.2: Placeholder scheduling.

There are any number of *execution hosts* (or *computational nodes*) – the machines on which the computations are actually executed. On each execution host there is one or more (usually depending on the number of CPUs) *placeholder*(s) running. A placeholder is a unit of potential work. It can be implemented as a shell script, or a script for a local batch scheduler, or in any other way. However, it has to use the service routines (or services) of the command-line server to perform the

following activities:

1. Get the next job from the server,

2. Execute the job, and

3. Resubmit itself.

In the simplest case, the placeholder can be implemented as a shell script containing only an infinite loop. Resubmission corresponds to looping around in this case. The placeholder stops when it gets information that there are no unexecuted jobs left.

```
While there are non-started jobs, do:
 1. Obtain a command line.
 2. Execute the command line.
 3. Signal completion.
```

Figure 2.3: Algorithm for a generic placeholder.

Basically, a placeholder executes the algorithm in Figure 2.3 [32]. A sample placeholder implemented as a PBS script is shown in Figure 2.4. The placeholder includes the ability to dynamically increase and decrease the number of placeholders in the PBS queue. The lines beginning with `#PBS` (lines 4-11, Figure 2.4) are directives interpreted by PBS at submission time. The command line is retrieved from the command-line server (in our case, using the program `getcmdline`) and stored into the `OPTIONS` shell variable (line 18, Figure 2.4). This variable is later evaluated at placeholder execution time with the command(s) that will be executed (line 50, Figure 2.4). The late binding of placeholder to executable name and command-line arguments is key to the flexibility of placeholder scheduling.

The placeholder then evaluates the amount of time it has been queuing for (line 32, Figure 2.4), and consults a local script to determine what action to take (line 38, Figure 2.4). It may increase the placeholders in the queue by one (lines 44-47, Figure 2.4), maintain the current number of placeholders in the queue by resubmitting itself after finishing the current command line (line 60, Figure 2.4), or decrease the

number of placeholders in the queue by not resubmitting itself after completing the current command line (lines 53-55, Figure 2.4).

```
1  #!/bin/sh
2  ## Generic placeholder PBS script
3
4  #PBS -S /bin/sh
5  #PBS -q queue
6  #PBS -l ncpus=4
7  #PBS -N Placeholder
8  #PBS -l walltime=02:00:00
9  #PBS -m ae
10 #PBS -M pinchak@cs.ualberta.ca
11 #PBS -j oe
12
13 ## Environment variables:
14 ##   CLS_MACHINE - points to the command-line server's host.
15 ##   CLS_DIR - remote directory in which the command-line server is located.
16 ##   ID_STR - information to pass to the command-line server.
17 ## Note the back-single-quote, which executes the quoted command.
18 OPTIONS=`ssh $CLS_MACHINE "$CLS_DIR/getcmdline $ID_STR"`
19
20 if [ $? -ne 0 ]; then
21     /bin/rm -f $HOME/MQ/$PBS_JOBID
22     exit 111
23 fi
24 if [ -z $OPTIONS ]; then
25     /bin/rm -f $HOME/MQ/$PBS_JOBID
26     exit 222
27 fi
28
29 STARTTIME=`cat $HOME/MQ/$PBS_JOBID`
30 NOWTIME=`$HOME/bin/mytime`
31 if [ -n "$STARTTIME" ] ; then
32     let DIFF=NOWTIME-STARTTIME
33 else
34     DIFF=-1
35 fi
36
37 ## Decide if we should increase, decrease, or maintain placeholders in the queue
38 WHATTODO=`$HOME/decide $DIFF`
39
40 if [ $WHATTODO = 'reduce' ] ; then
41     /bin/rm -f $HOME/MQ/$PBS_JOBID
42 fi
43
44 if [ $WHATTODO = 'increase' ]; then
45     NEWJOBID=`/usr/bin/qsub $HOME/psrs/aurora-pj.pbs`
46     $HOME/bin/mytime > $HOME/MQ/$NEWJOBID
47 fi
48
49 ## Execute the command from the command-line server
50 $OPTIONS
51
52 ## leave if 'reduce'
53 if [ $WHATTODO = 'reduce' ] ; then
54     exit 0
55 fi
56
57 /bin/rm -f $HOME/MQ/$PBS_JOBID
58
59 ## Recreate ourselves if 'maintain' or 'increase'
60 NEWJOBID=`/usr/bin/qsub $HOME/psrs/aurora-pj.pbs`
61
62 $HOME/bin/mytime > $HOME/MQ/$NEWJOBID
```

Figure 2.4: Generic PBS placeholder [32].

2.4 Grids

We adopt the widely-accepted definition of a grid by Ian Foster [9]. According to that definition, a grid "*is a system that:*

1. *coordinates resources that are not subject to centralized control*
2. *using standard, open, general-purpose protocols and interfaces*
3. *to deliver nontrivial qualities of service.*"

Some metacomputing systems come close to satisfying this definition. However, most of them fail to use "standard, open, general-purpose protocols and interfaces." In particular, Trellis is not a grid, because it does not satisfy the second and the third conditions of the above definition.

2.4.1 Legion

In order to understand the main principle of Legion[3] [20, 21, 22], we look back at the concept of an operating system (OS) that the reader is certainly familiar with. According to one definition [28], "*an OS is a set of software elements that turn the physical resources constituting a computing system into a collection of virtual objects, with the intention of providing a convenient and efficient virtual platform for the development, execution, and maintenance of programs, as well as for information storage and retrieval*". By this definition, Legion is an operating system for the computing system that represents a possibly large collection of decentralized computational resources.

Legion defines an object-oriented model in which all the resources are objects that are instances of corresponding classes. Examples of Legion objects include: a host, a file residing on some host, a printer, etc. Legion does not specify a particular implementation of its classes' interfaces; thus, it provides a general-purpose framework for creating non-trivial applications that harness the power of the available computational resources.

[3] In the Roman empire, legion was an army unit of 3,000-6,000 infantry and 100-200 cavalry.

Legion meets only two of the three conditions of our definition of a grid: its interfaces have not gone through a rigorous standardization process that Ian Foster required.

As powerful a platform as it is, Legion has one major drawback: in order to benefit from Legion's great features, an application has to be Legion-aware, i.e. it has to be reimplemented to connect to Legion's object-oriented interfaces. In contrast, TrellisDAG does not impose any such requirements on an application.

2.4.2 Globus

Globus [13, 10] is a grid toolkit that started out in the Argonne National Laboratory and since became a multi-institutional project with significant industry support. It represents a set of software components that any application can use to take advantage of the resources visible to the toolkit. There are two major differences between the Legion (see Section 2.4.1) project and the Globus project:

1. Legion emphasizes an object-oriented programming model; Globus applications can use any programming model, and

2. Whenever possible, the developers of Globus use existing standards (e.g. LDAP – Lightweight Directory Access Protocol); furthermore, they aim to augment the standards by the Globus protocols and interfaces.

The Globus project is ambitious and wants to address all the problems of grid computing including security, status monitors, and automatic resource discovery. We will describe parts of the toolkit most relevant to this thesis (refer to Figure 2.5):

1. **The Globus Resource Allocation Manager (GRAM)**. This software component is a bottom level of the hierarchy of components that provide global resource allocation. It is responsible for managing local resources. For example, there could be an installation of LSF on a group of computers. Then there would be a GRAM component that managed this group of computers. Thus, GRAM components work on top of the local resource management systems and bridge them to the global services of the toolkit.

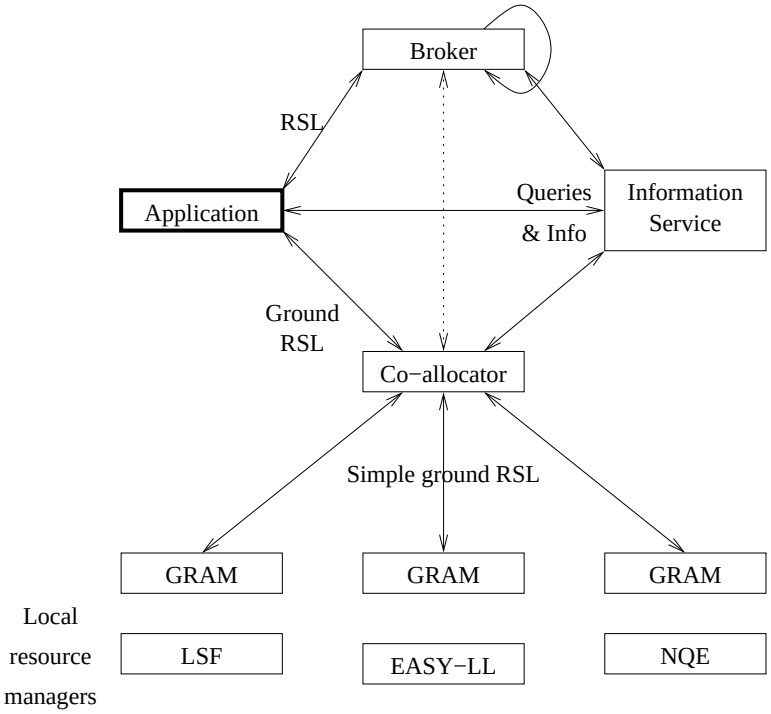

Figure 2.5: Resource management with Globus (reproduced from Foster and Kesselman [10]).

The global resource allocation works as follows. When an application needs a resource, it specifies the need in a special language called the Resource Specification Language (RSL). This specification is passed to a component called the *broker*, which identifies a subset of GRAMs that fit the description and refines the specification accordingly. The resulting specification can be passed to another broker for further refinement. Eventually a set of GRAMs that best match the specification is found. Then the specification is broken into a set of specific requests that are passed to the GRAM components.

It is possible that resources corresponding to several GRAMs have to be allocated simultaneously or *co-allocated*. Therefore, there are global components called co-allocators built on top of several GRAM components. These components receive the resource requests and co-allocate GRAMS as necessary.

2. **The Globus Metacomputing Directory Service (MDS)**. A computational grid is a dynamic system, in which new resources may be added, other re-

sources removed, and some resources changed. The grid components and applications need to have means to get information about various components (both software and hardware) in order to adapt to the changes.

MDS provides interfaces for publishing information about the state of the grid as well as for accessing that information. It uses the Lightweight Directory Access Protocol (LDAP) for representing the information.

Other components and applications have a responsibility to use the publishing interfaces of MDS to post information about their status.

3. **The Global Access to Secondary Storage (GASS).** The idea is somewhat similar to the Trellis File System's wrappers. The standard I/O functions are overridden in order to provide remote access functionality.

Several grids have been constructed using the Globus toolkit, including:

1. Grid Physics Network (GriPhyN) [14] – a grid for collecting and analyzing scientific and engineering data on large scale. There are several other projects with similar data-oriented goals [6, 16, 29, 40],

2. Information Power Grid [27] – an ambitious grid project by NASA with the goal of building an all-purposes grid with advanced services on top of the Globus toolkit, and

3. TeraGrid [39] – a project with the main goal of creating a grid-based computing research environment for several American universities. A project with similar goals exists in Denmark [7].

Most grids built with the Globus technology use the DAGMan tool of Condor to tackle the problem of job dependencies.

Work is under progress to create a standard called the *Open Grid Services Architecture* based on the Globus toolkit [11, 12].

2.5 Policies

In the previous sections of this chapter, we surveyed several systems that provide mechanisms for harnessing computational resources. Usually, these mechanisms are generic enough to allow for many ways of mapping individual jobs to resources; such is TrellisDAG, whose mechanisms are generic. In this section, we introduce standard terminology used in the HPC community to discuss the issues of scheduling. In particular, we are interested in scheduling in the context of workflows with inter-job dependencies.

Assume a job that has to be executed using TrellisDAG or any other system such as the ones described in this survey. The *lifetime* of a job can be conceptually split into stages, as shown in Figure 2.6.

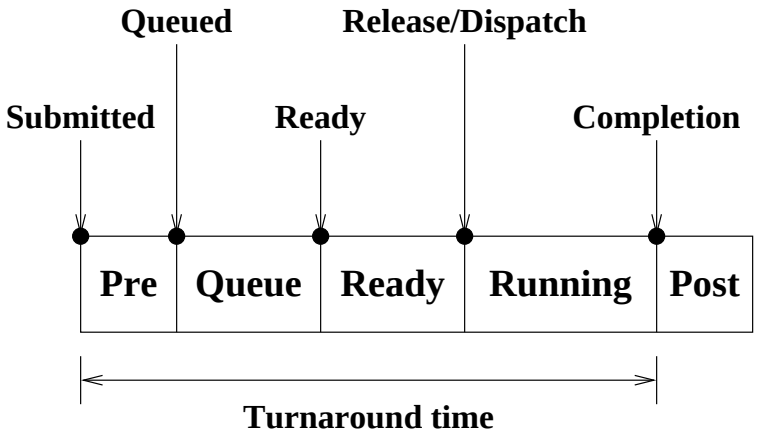

Figure 2.6: Stages of the lifetime of a job.

We discuss the stages in the order from left to right on this figure. First, the job has to be submitted to the system. The moment of submission is called the *submission time* of the job (labeled as "Submitted" in Figure 2.6). The submitted job is entered into the *pre-queue* (labeled as "Pre" in Figure 2.6). The job stays in the pre-queue until all of its *static dependencies* are satisfied. Static dependencies are the ones that do not depend on the result of execution of other jobs. For example, compilation of the executable for the job may be a static dependency.

Once all the static dependencies are satisfied, the job is entered into the *queue*. A job stays in the queue as long as there are unmet dynamic (or workflow) depen-

dencies.

When all of the dynamic dependencies of the job are satisfied, it is entered into the *ready queue*. The moment when this happens is called the ready time of the job. The job stays in the ready queue until the system is able to allocate a physical resource and run the job, at which point the state of the job is changed to *running*. This moment of time is called the *release (or dispatch) time* of the job.

The job is completed at *completion time* and is entered into the *post-queue* (labeled as "Post" in Figure 2.6), thereby allowing the system to perform post-tasks, such as, for example, removal of the executable of the job.

Now we introduce some more terminology that is used to describe scheduling engines in HPC environments. The terminology comes from a well-known paper by Feitelson, Rudolph, Schwiegelshohn, Sevcik and Wong [8].

Suppose that a set of jobs J has been submitted to the environment. Then, with every job $j \in J$, we can associate the following parameters:

1. s_j – the release time,
2. t_j – the completion time, and
3. w_j – the priority level of j.

The following cost metrics are commonly used to evaluate a schedule:

1. $\max_{j \in J} t_j$ – *makespan* or *throughput*. Note that usually throughput is defined as the number of jobs (or, in this context, the sum of priorities of jobs) completed per time unit. We will use the term in the latter sense,
2. $\sum_{j \in J} w_j t_j$ – weighted completion time, and
3. $\sum_{j \in J} w_j (t_j - s_j)$ – weighted response time.

Jobs are usually matched to a *partition* (i.e., set of resources) of the HPC environment according to the job's specifications (such as required number of processors and amount of memory) and the state of the environment (such as the workload). Depending on how the partition given to a job can or cannot change, the partitions can be *fixed* (i.e. determined at submission time (refer to Figure 2.6) by

the administrator), *variable* (i.e. chosen at submission time according to user requirements), *adaptive* (i.e. chosen at release time by the environment), or *dynamic* (i.e. can change during the execution).

Preemption adds another dimension to the choices that the scheduler can make in order to obtain a better schedule. It is useful to classify the jobs according to the degree of preemption that they support. A job may support no preemption; alternatively, it can support *local preemption* (i.e. threads of the job can be preempted, but must be later resumed on the same processor), *migratable preemption* (i.e. threads of the job can be preempted and later resumed on a different processor), or *gang scheduling* (i.e. all threads of the job are preempted and restarted simultaneously, either with or without migration).

Given the cost metrics for schedules and the classification of jobs according to their flexibility and support of preemption, the scheduler has to come up with a schedule.

One of the main characteristics of a scheduler is what information is available to it and what information it can use for decision making. Four main cases are distinguished:

1. The scheduler does not have/use any knowledge,
2. The scheduler maintains information about workload and its distribution,
3. The scheduler classifies jobs according to several important properties (e.g. estimated parallelism) and uses the information about the **class** of the individual jobs, or
4. The scheduler uses estimates of execution time of individual jobs (each job may have many estimates depending on partition size given).

We proceed to introduce additional terminology that is used to describe scheduling issues for workflows with dependencies. Our source is a well-known paper by Kwok and Ahmad [17].

The set of jobs can be modeled as a directed acyclic graph (DAG) with a set of vertices (or nodes) V and a set of arcs E. Each node represents a single job. If

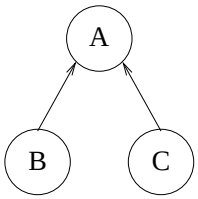

Figure 2.7: A simple workflow example.

$v_1, v_2 \in V$ and there is an arc $v_2v_1 \in E$, then the job v_2 cannot be executed until the job v_1 is completed and its results are available. For example, job C and job A are examples of such jobs v_2 and v_1, respectively, in Figure 2.7. v_1 is called the *parent node* and v_2 is called the *child node*. A node with no parent is an *entry node* and a node with no child is an *exit node*. Each DAG has one or more entry nodes and one or more exit nodes. For example, the DAG in Figure 2.7 has one entry node A and two exit nodes B and C. We assume that all jobs are sequential and are executed on one processor. In this thesis, we do not deal with preemption (i.e. we assume the run-to-completion policy).

We further associate a weight with every node of the DAG. The weight associated with a node $v \in V$ is denoted by $w(v)$ and represents the expected execution time of v. We also associate a weight with every arc and these weights are called $costs$. The cost associated with the arc $v_2v_1 \in E$ is the overhead of transferring the data generated by v_1 in order to start v_2. This cost is denoted by $c(v_1, v_2)$.

Scheduling jobs of a DAG as described above to a limited number of processors so as to minimize the makespan is an NP-complete problem. Many heuristic approaches to scheduling have been proposed (see [17] for details).

2.6 Concluding remarks

Harnessing the available computational resources has been a subject of efforts of the high-performance computing community for years. As a result of these efforts, we have systems ranging in sophistication from simple batch schedulers to grids. However, there is no single system that is both easy to install and use and covers all the needs of the computational science community. In this respect, TrellisDAG's

interesting feature is that it can work on top of other systems without sacrificing any of its distinct features related to handling of inter-job dependencies.

Chapter 3

The Motivating Application

First we introduce the motivating application – building checkers endgame databases via retrograde analysis – at a very high level. Then we highlight the properties of the application that are important for our project. Finally, we argue that many scientific applications exhibit these properties and it is therefore desirable to have a system that is able to take advantage of these properties. We used the checkers databases as one example of an application, but the design of TrellisDAG does not make any application-specific assumptions and can be applied to other application domains.

3.1 Introducing the application

A team of researchers in the Department of Computing Science at the University of Alberta aims to solve the game of checkers [18]. The rules of the game of checkers are briefly summarized in Appendix C. For any given position the following question has to be answered:

Can the side to move first force a win or a draw?

In order to answer this question, the investigators of the checkers project combine retrograde analysis [19] and forward search. During the retrograde analysis, a database of endgame positions is constructed. Each entry in the database corresponds to a unique position and contains one of 3 values: WIN, $LOSS$ or $DRAW$. Such a value represents perfect information about a position and is called the *theoretical value* for that position.

The construction starts with the trivial case of one piece. We know that whoever has that last piece is the winner. In general, given a position, whenever there is at least one legal move that leads to a position that has already been entered in the database as a $LOSS$ for the opponent side, we know that the given position is a WIN for the side to move (since he can take that move, the values of all other moves do not matter); conversely, if all legal moves lead to positions that were entered as a WIN for the opponent side, we know that the given position is a $LOSS$ for the side to move. For a selected part of databases, this analysis goes in iterations. An iteration consists of going through all positions to which no value has been assigned and trying to derive a value using the rules described above. When an iteration does not result in any updates of values, then the remaining positions are assigned a value of $DRAW$ since neither player can force a win.

If we could continue the process of retrograde analysis up to the initial position with 24 pieces on the board, then we would have solved the game. However, the number of positions grows exponentially and there are on the order of 10^{20} possible positions in the game. That is why retrograde analysis is combined with a forward search, in which a game tree with the root as the initial position of the game is constructed. When the two approaches "meet", we will have perfect information about the initial position of the game and the game will be solved.

The program that we use for this project implements the retrograde analysis approach [19].

3.1.1 Important application-specific characteristics

Division of Databases

Early on, the researchers of the checkers project saw that there were opportunities for massive parallelism in the computation of checkers endgame databases [19]. This was one of the reasons why the checkers databases were divided into smaller and smaller parts; the other reason was to reduce the memory footprint – the amount of memory used at any one time. We will now describe the logic behind this division as well as the opportunities for concurrency.

The levels of subdivision are summarized in Figure 3.1. Our explanation follows

this diagram in the direction from top to bottom.

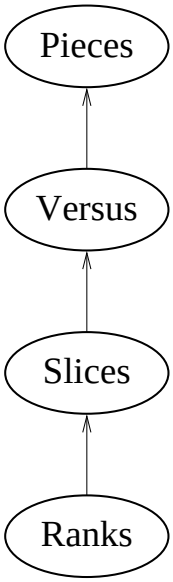

Figure 3.1: Levels of subdivision of checkers databases.

At the highest level, it is natural to divide the set of all possible positions into subsets according to the total number of pieces on the board as shown in Figure 3.2. The least non-trivial number of pieces is 2. Given the size of the databases, we cannot expect to compute the 11-piece databases any time soon. For now we assume a simple model of retrograde analysis where the databases for $n + 1$ pieces can only start being computed when the complete databases for n pieces have been obtained. Figure 3.2 shows the workflow dependencies so far.

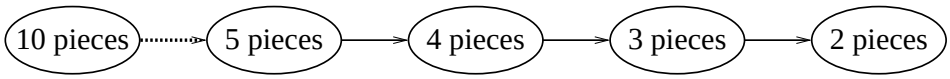

Figure 3.2: Workflow dependencies at the level of pieces.

Assume that that the two players are black and white. The following observation is important: the theoretical value of two distinct positions can be computed in parallel (i.e. independently) if and only if neither of these positions can reach the other by a sequence of legal moves. For example, consider a position with 5 pieces on the board in which black has 2 pieces and white has 3 pieces, and another

position with the same number of pieces in which black has only 1 piece and white has 4 pieces. These two positions can be processed in parallel because there is no way for one of them to result in the other in any game of checkers, since the number of pieces of neither color can increase during the game.

At the next level, we further subdivide the databases according to the number of pieces that each side has on the board. For example, for 7-piece databases, there are three independent subsets: 4 pieces versus 3 pieces, 5 pieces versus 2 pieces, and 6 pieces versus 1 piece. Note that two positions in any two different groups listed above can be processed in parallel by the above observation. Therefore, if we expand the node in Figure 3.2 corresponding to 7-piece databases, then we have no workflow dependencies for computing 7-piece databases (see Figure 3.3).

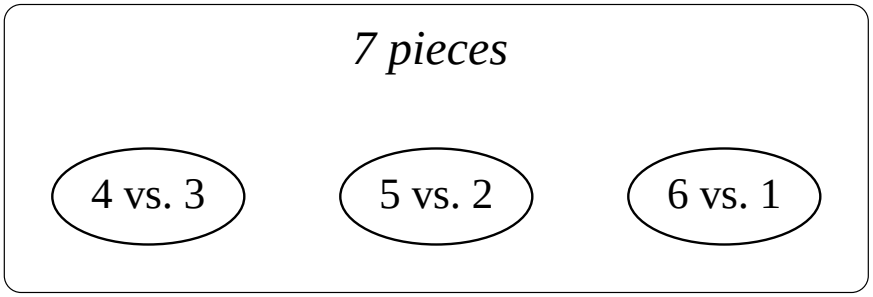

Figure 3.3: Workflow dependencies at the level of pieces on each side (i.e. the `versus` level).

We can subdivide the databases further. Suppose that black has 4 pieces and white has 3 pieces. We note that a position with 2 black kings, 2 black checkers, and 3 white kings can never play into a position with 3 black kings, 1 black checker, 2 white kings, and 1 white checker or vice versa – because the change in the number of black kings corresponds to the forward direction in the game and the change in the number of white kings corresponds to the backwards direction. Thus, any two positions from those two categories can be processed in parallel.

In general, let us denote any position by four numbers standing for the number of kings and checkers of each color on the board. For example, `2113` is read from left to right as follows: a position with 2 black kings, 1 white king, 1 black checker, and 3 white checkers. Such a position is shown in Figure 3.4(a). We can

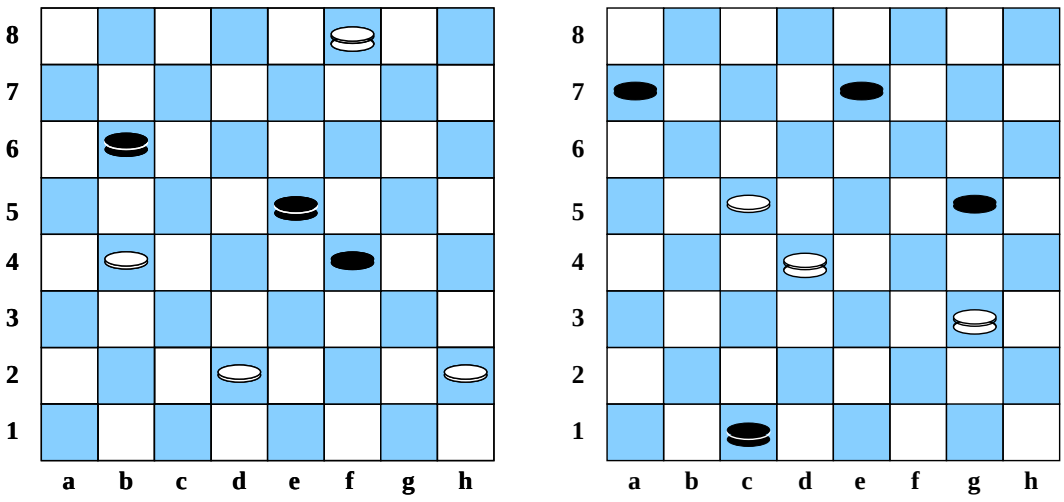

Figure 3.4: Positions from the sets `2113` and `1231`.

always assume that black is to move because we will also consider the symmetrical positions from the group `1231`. For example, Figure 3.4(b) shows the position from the group `1231` that corresponds to the position in Figure 3.4(a).

A group of all positions that have the same 4-number representation is called a *slice*.

Positions $b_k^1 w_k^1 b_c^1 w_c^1$ and $b_k^2 w_k^2 b_c^2 w_c^2$ that have the same number of black and white pieces can be processed in parallel if one of the following two conditions hold:

$$w_k^1 > w_k^2 \text{ and } b_k^1 < b_k^2$$

or

$$w_k^1 < w_k^2 \text{ and } b_k^1 > b_k^2.$$

Figure 3.5 shows the workflow dependencies for the case of 4 pieces versus 3 pieces.

Each of the slices in Figure 3.5 supports additional concurrency. In order to see that, let us order the positions that have the same number of checkers and kings of each color by the *rank* of the leading checker. The rank of a checker is defined as follows:

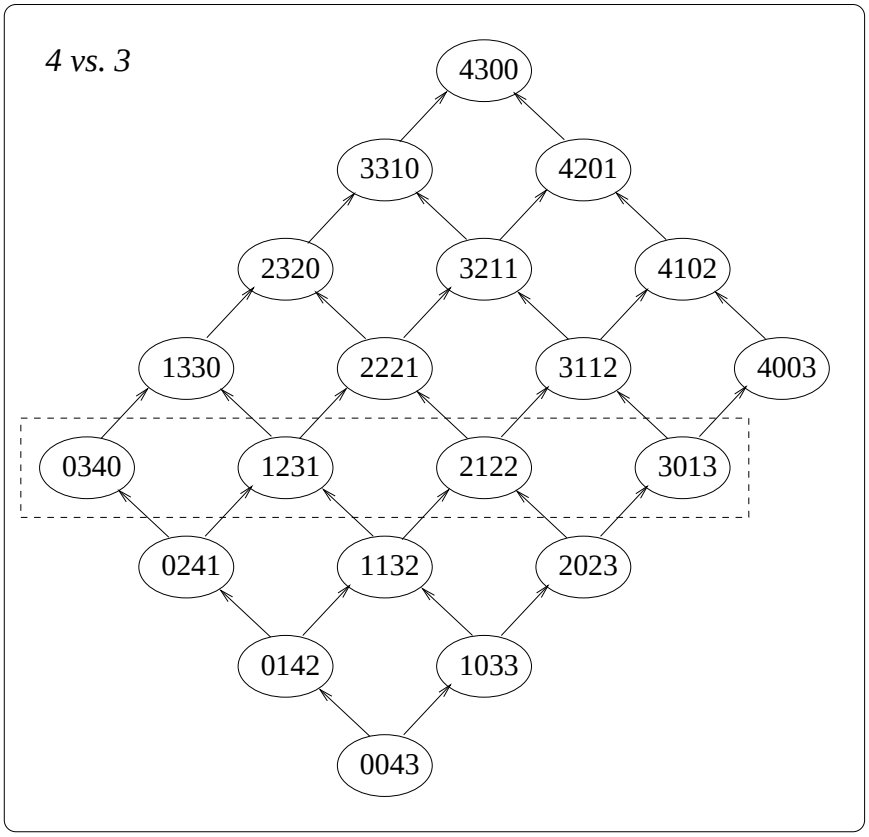

Figure 3.5: Workflow dependencies at the level of slices.

1. The rank of a black checker is counted from 0 starting from line 8 of the board. For example, the leading black checker in Figure 3.4(a) is on line 4, which is rank 5 and

2. The rank of a white checker is counted from 0 starting from line 1 of the board. For example, the leading white checker in Figure 3.4(a) is on line 4, which is rank 3.

Thus ranks are numbers from 0 to 6 (on rank 7 a checker would become a king). Let $b_l w_l$ refer to a position with the leading checker for black on line b_l and the leading checker for white on line w_l. Then a part of databases at the finest level of subdivision is denoted by 6 numbers as $b_k w_k b_c w_c . b_l w_l$. For example, the position in Figure 3.4(a) is from the group of positions `2113.53`. The workflow dependencies at the level of ranks are illustrated in Figure 3.6 for three cases (from left to right):

1. Both sides have checkers,

2. Only black has checkers, and

3. Only white has checkers.

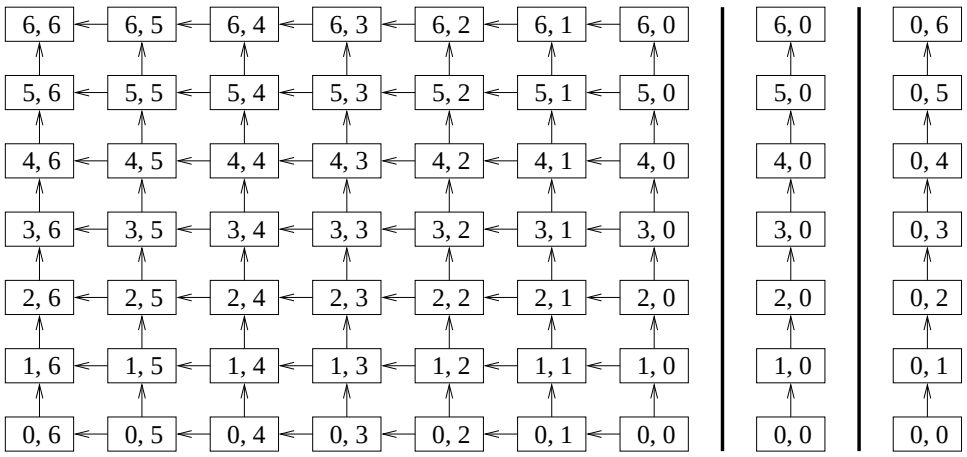

Figure 3.6: Workflow dependencies at the level of ranks.

We emphasize that the workflow dependencies emerge from:

1. The rules of checkers,

2. The retrograde analysis algorithm, and

3. The subdivision of the checkers endgame databases.

TrellisDAG does not contribute any workflow dependencies in addition to those listed above.

Stages for computing a part of the checkers endgame databases

Several stages[1] are executed for each part of the databases at the level of ranks:

1. Computation. At this stage the positions of the part are assigned their $WIN/LOSS/DRAW$ value and this value is stored in the database for the part being computed. The stage uses the complete *compressed* (see below) databases computed to date,

[1]The reader is referred to Lake, Schaeffer, Lu [19] for a more detailed description of the computation.

2. Compression. The databases are compressed to save disk space,

3. Addition. The compressed databases are added to the complete databases, and

4. Verification. The complete databases are checked for consistency.

3.2 Properties of the checkers application with respect to TrellisDAG

Although building the checkers endgame databases motivated our work on TrellisDAG, the system can handle a wide class of applications with or without inherent workflow dependencies. Building the checkers endgame databases is just one of many possible applications for TrellisDAG. In this section, we summarize the properties of the checkers application that make it well suited for testing TrellisDAG. We use computing the part of the 7-piece databases for 4 pieces versus 3 pieces (see Figures 3.5 and 3.6) as a running example for demonstrating these properties.

1. Variable degree of concurrency. The computation of 4 vs. 3 slice starts with computing the databases for the kings only situation and admits no concurrency. However, it can happen that slices `0340`, `1231`, `2122`, and `3013` (the area marked by the dashed rectangle in Figure 3.5) are being computed simultaneously; moreover, in each of the latter three slices, there is a possibility of 7 jobs computing concurrently (corresponding to the up-diagonal on the left part of Figure 3.6). Thus, the maximal degree of concurrency is $7 \times 3 + 1 = 22$.

2. Varying length of jobs. On a 1.5 GHz Athlon with 512 MB of memory, computing the `4300` slice takes more than an hour, while some of the shorter jobs take as little as 6 seconds.

3. Large number of jobs. Computing the 4 vs. 3 slice of the databases involves 638 jobs. In our experiments, we separate the verification stage into a separate job, leading to a total of $2 \times 638 = 1,276$ jobs.

4. Non-trivial workflow dependencies.

3.3 Concluding remarks

Computing checkers endgame databases is a non-trivial application that requires large computing capacity and presents a challenge by demanding several properties of a metacomputing system:

1. A tool/mechanism for convenient description of a multitude of jobs and inter-job dependencies,

2. The ability to dynamically satisfy inter-job dependencies while efficiently using the application-specific opportunities for concurrent execution, and

3. A tool for convenient monitoring and administration of the computation.

TrellisDAG prototypes a system that combines the technique of placeholder scheduling with a hierarchical subsystem for describing a workflow to provide for these properties. In the next chapter, we look at the features of TrellisDAG in detail.

Chapter 4

Overview of TrellisDAG

An architectural view of TrellisDAG is presented in Figure 4.1. We start by introducing the running example for our discussion in Section 4.1. Simultaneously, we introduce the group model – a concept that facilitates convenient handling of complicated workflows (such as the checkers computation [19]) and provides a mechanism for other developments (see Sections 1.1 and 7.1). To run a computation, the user has to submit the jobs and their dependencies to the *jobs database* using one of the several methods described in Section 4.2. Then one or more placeholders have to be run on the execution nodes (workstations). These placeholders use the services of the command-line server to access and modify the jobs database. The command-line server is described in Section 4.3. Finally, the monitoring and administrative utilities are described in Section 4.4.

4.1 The group model and the running example

In our system, each job is part of a *group* of jobs and explicit dependencies exist between groups rather than between individual jobs. This simplifies the dependencies between jobs (i.e. the order of execution of jobs within a group is determined by the order of their submission).

A group may contain either only jobs or both jobs and *subgroups*. A group is called the *supergroup* with respect to its subgroups. A group that does not have subgroups is said to be a group *at the lowest level*. In contrast, a group that does not have a supergroup is said to be a group *at the highest level*. In general, we say that

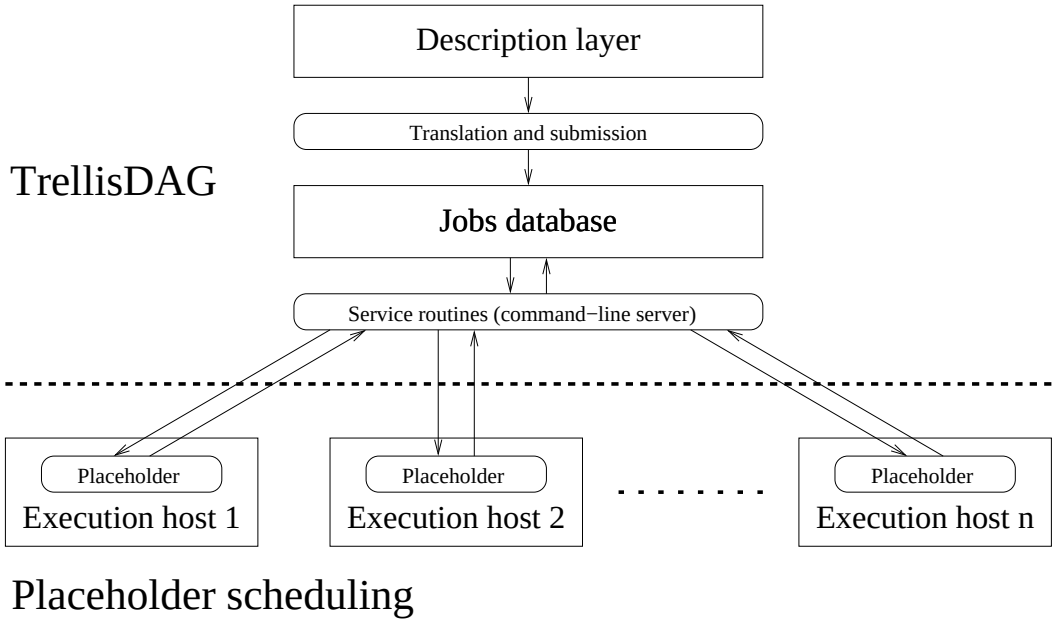

Figure 4.1: An architectural view of TrellisDAG.

a subgroup is one level lower than its immediate supergroup.

With each group, there is associated a special group called the *prologue group*. The prologue group logically belongs to the level of its group, but it does not have any subgroups. Jobs of the prologue group (called *prologue jobs*) are executed before any job of the subgroups of the group is executed.

We also distinguish *epilogue jobs*. In contrast to prologue jobs, epilogue jobs are executed after all other jobs of the group are complete. In this version of the system, epilogue jobs of a group are part of that group and do not form a separate group (unlike the prologue jobs).

Note the following:

1. Jobs within a group will be executed in the order of their submission. In effect, they represent a pipeline. This is illustrated in Figure 4.2.

2. Dependencies can only be specified between groups and never between individual jobs. If such a dependency is required, a group can always be defined to contain one job. This is illustrated in Figure 4.3.

3. A supergroup may have jobs of its own. Such jobs are executed after all

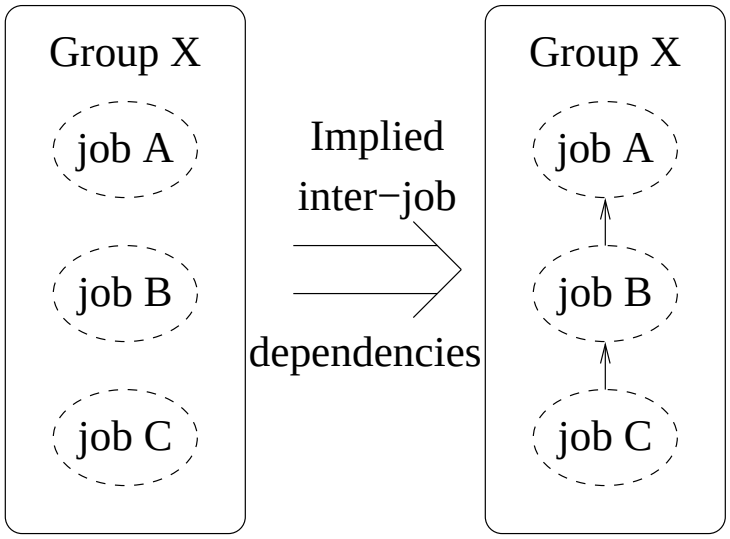

Figure 4.2: Dashed ovals denote jobs. The dependencies between the jobs of group X are implicit and determined by the order of their submission.

subgroups of the supergroup are completed. This is illustrated in Figure 4.4.

4. Dependencies can only be defined between groups with the same supergroup or between groups at the highest level (i.e. the groups that do not have a supergroup). This is illustrated in Figure 4.5.

5. Dependencies between supergroups imply pairwise dependencies between their subgroups. These extra dependencies do not make the workflow incorrect, but are an important consideration, since they may inhibit the use of concurrency. This is illustrated in Figure 4.6.

Assume that we have the 2-piece databases computed and that we would like to compute the 3-piece and 4-piece databases. We assume that there are scripts to compute individual slices. For example, running script `2100.sh` would compute and verify the databases for all positions with 2 kings of one color and 1 king of the other color (in effect, a script $b_k w_k b_c w_c$`.sh` computes and verifies two slices: $b_k w_k b_c w_c$ and $w_k b_k w_c b_c$, see Section 3.1.1). The workflow for our example is shown in Figure 4.7. The group `2001` in Figure 4.7 contains a single job (`2001.sh`) and can only be executed after the group `2100` is complete.

We can think of defining supergroups for the workflow in Figure 4.7 as shown

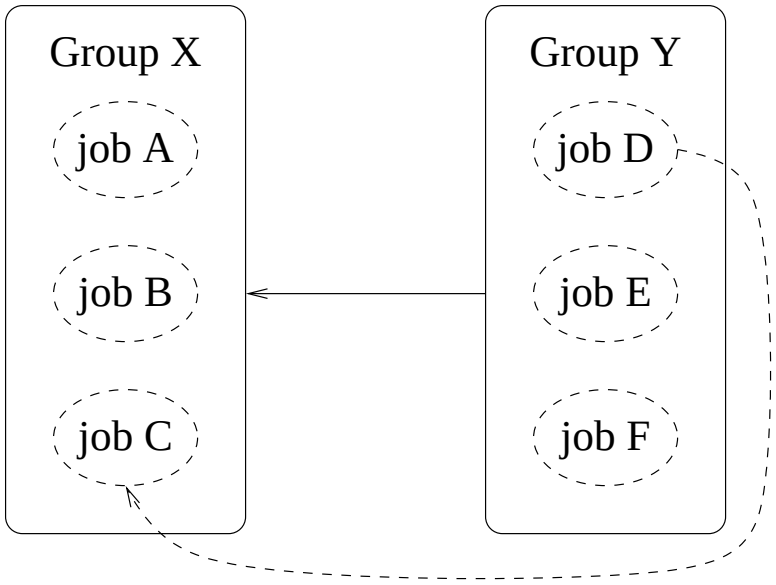

Figure 4.3: The workflow dependency of group Y on group X implies the dependency of the first job of Y on the last job of X (this dependency is shown by the dashed arc).

by the dashed lines. Then, we can define dependencies between the supergroups and obtain a simpler looking workflow in Figure 4.8.

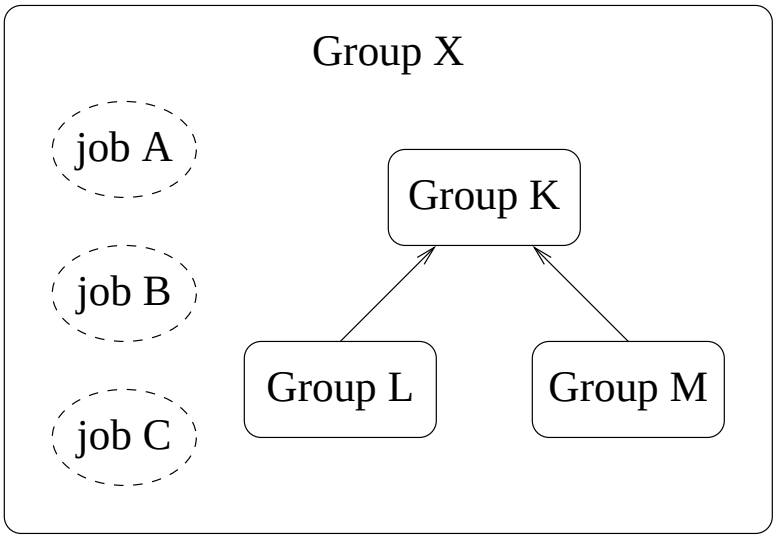

Figure 4.4: Group X has subgroups K, L, M and jobs A, B, C. These jobs will be executed after all the jobs of K, L, M and their subgroups are completed.

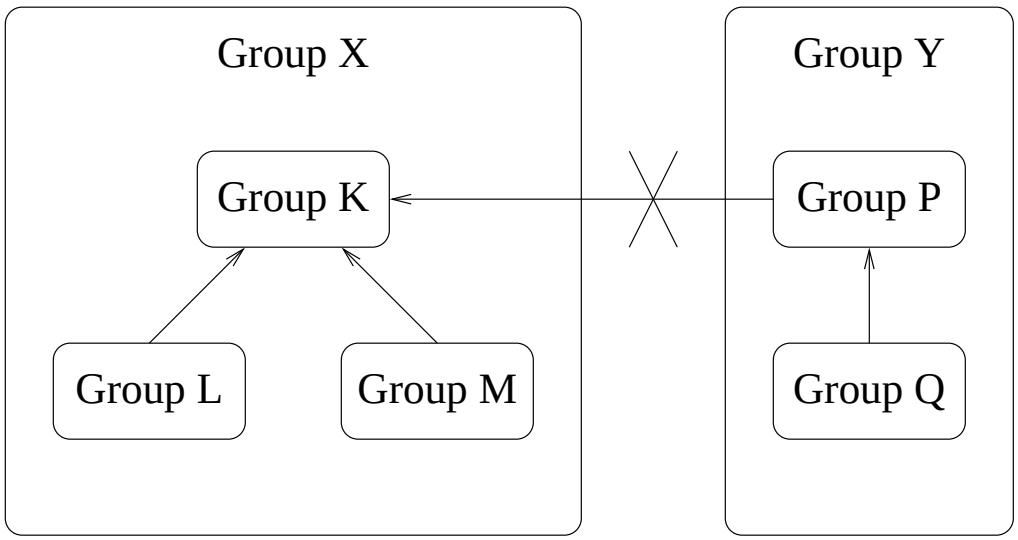

Figure 4.5: Groups K, L, M have same supergroup X; therefore, we can specify dependencies between these groups. Similarly, group Y is a common supergroup for groups P and Q. In contrast, groups K and P do not have a common supergroup (at least not immediate supergroup) and cannot have an explicit dependency between them.

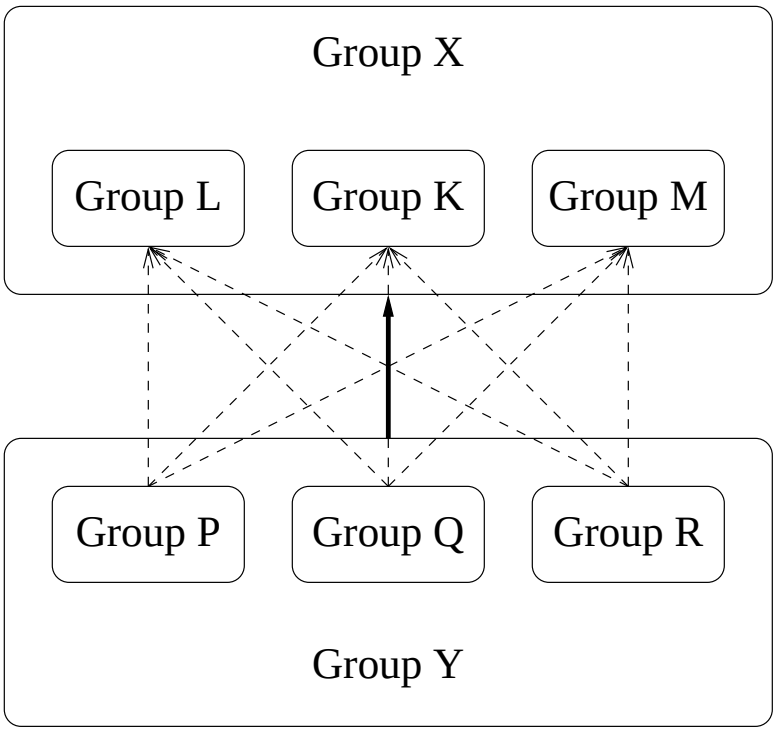

Figure 4.6: Group Y depends on group X and this implies pairwise dependencies between their subgroups (these dependencies are denoted by dashed arrows).

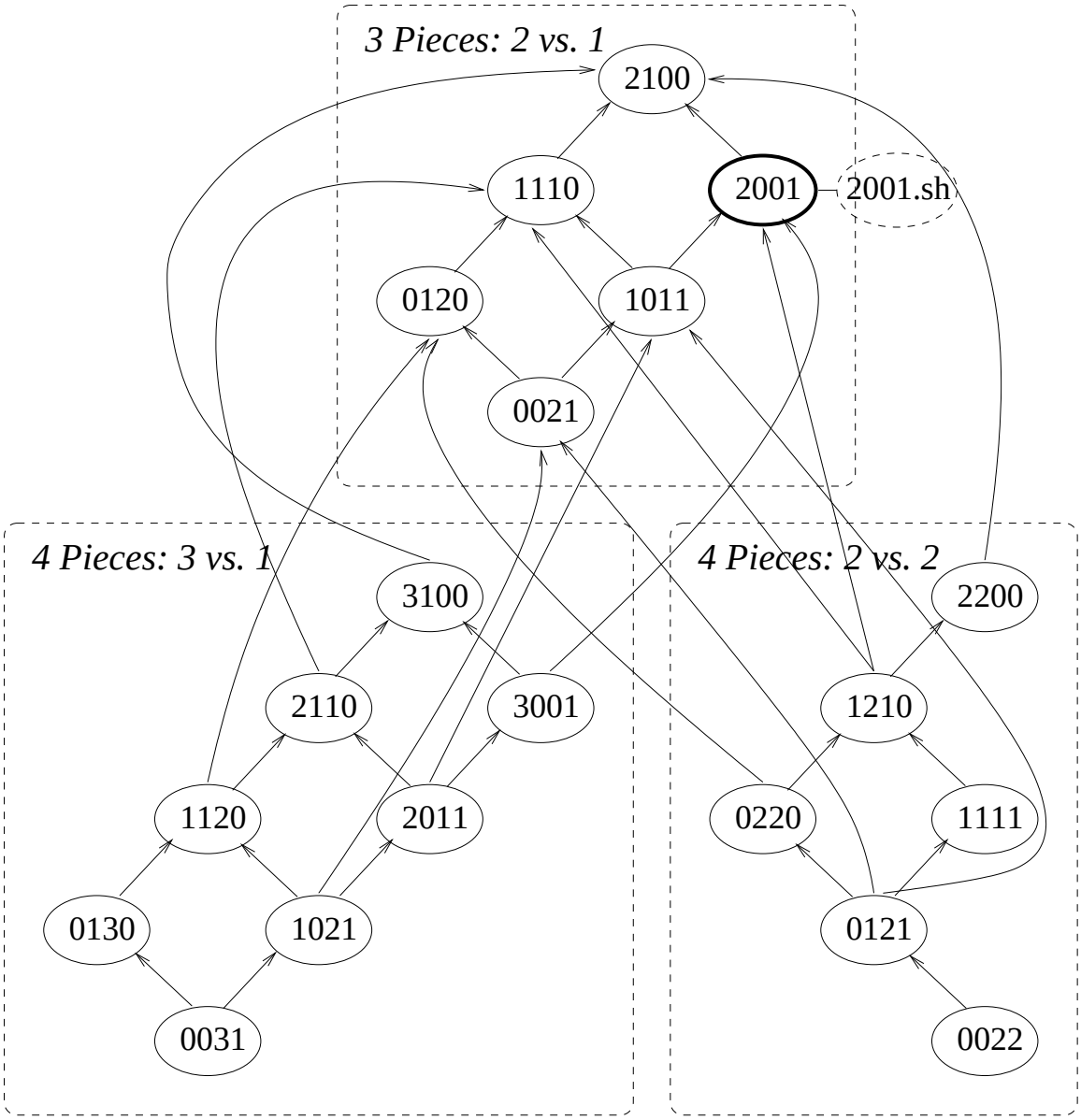

Figure 4.7: The workflow for the running example. The nodes are slices. Here slices are groups at the lowest and highest levels at the same time, since there is only one level. The figure shows how the slice `2001` contains only one job: `2001.sh`.

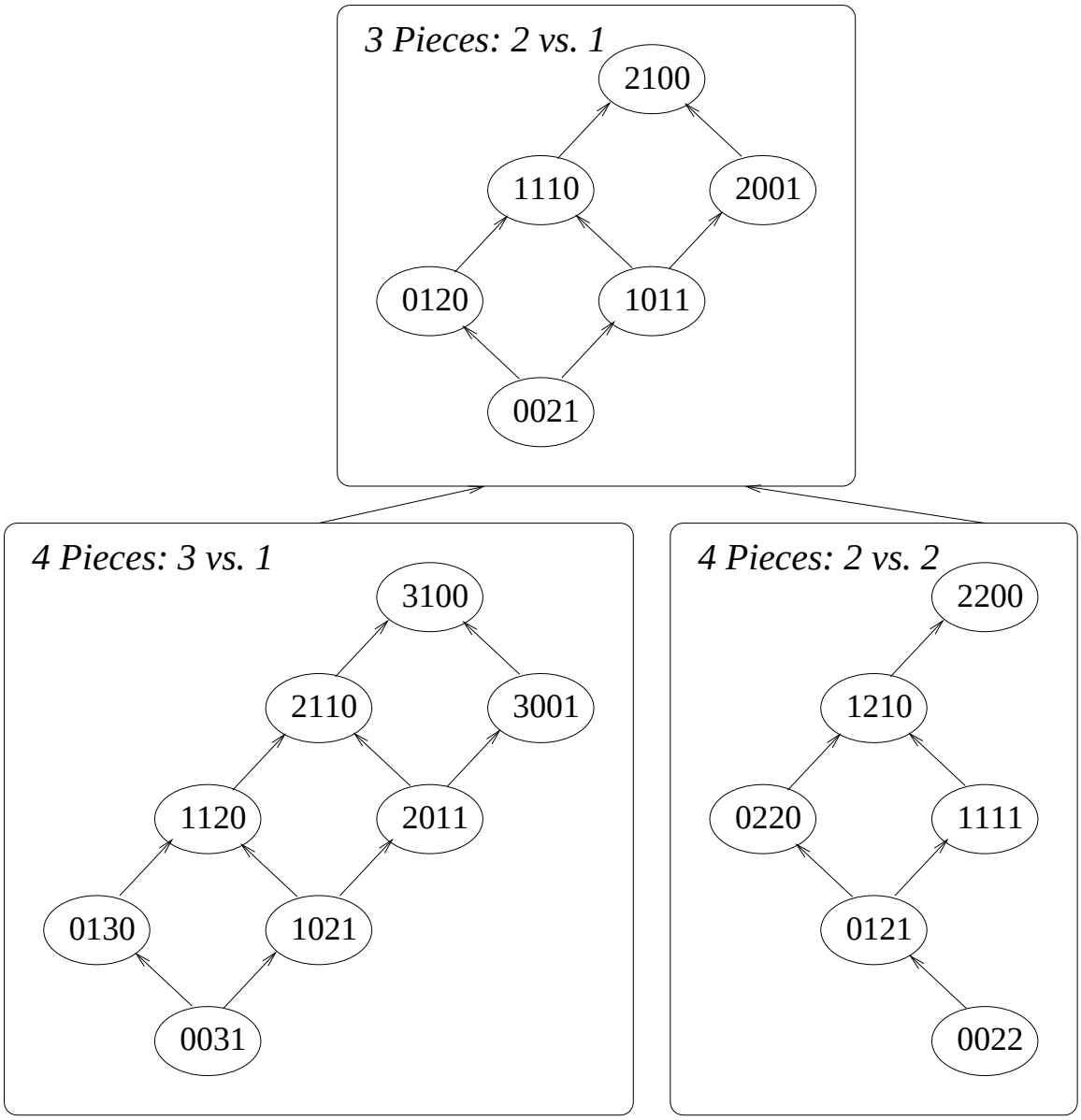

Figure 4.8: The workflow for the running example with supergroups. The level `versus` is the highest and groups at this level are supergroups for slices that are groups at the lowest level in this example.

Command/ Parameter	Description
mqsub	Submit one job to the jobs database
-l	Name of the group
-deps	Names of the groups on which the group depends
-comm	The command line of the job
-attr	Job's attributes

Table 4.1: Short description of the `mqsub` utility.

4.2 Submitting the workflow

The submission procedure is shown in Figure 4.9. The figure shows that there are several ways of describing the workflow. The user chooses the way depending on how complicated the workflow is and how he wants (or does not want) to make use of the grouping capability of the system.

Whatever way of describing the workflow is chosen, the description is translated into the form of the jobs database (see Figure 4.1) – a relational database that contains both static and dynamic information about the jobs and dependencies (see Section 5.2 for more details about the format of the jobs database).

We start with the simplest tools and work our way up to the most advanced way that allows definition of supergroups (this roughly corresponds to the order from the bottom upwards in Figure 4.9).

The utility for submitting a single job of a given group to the jobs database is called `mqsub`. The short description of `mqsub` is given in Table 4.1. There is no necessity to create groups explicitly. Whenever a group does not exist, TrellisDAG will create it automatically.

There are two limitations on the order of submission of the constituents of the workflow to the system:

1. The groups have to be submitted in some legal (with respect to workflow dependencies) order, and

2. The jobs within a group have to be submitted in the correct order.

The first of these limitations is taken care of by TrellisDAG when we use a DAG

description script for submission (see Section 4.2.2). The second limitation is intentional, because it allows for an implicit assumption about the pipeline dependencies between the jobs of a group; the user does not need to worry about specifying such dependencies.

In our example, we define a group for each slice to be computed. The script in Figure 4.10 submits the workflow in Figure 4.7.

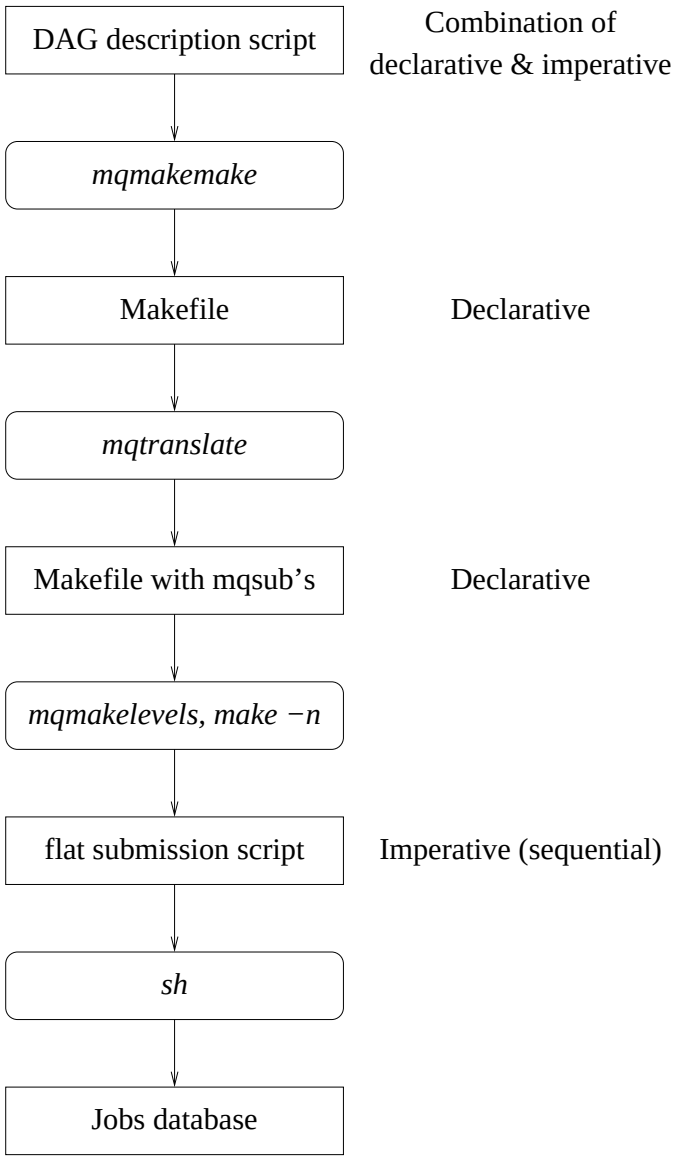

Figure 4.9: The complete submission procedure. Next to each form of description of the workflow is its corresponding programming paradigm with respect to submitting the workflow to the jobs database. Note that, with respect to executing the jobs, all of the forms of description are declarative.

```
mqsub -deps ""               -l "2100" -c "2100.sh"
mqsub -deps "2100"           -l "1110" -c "1110.sh"
mqsub -deps "1110"           -l "0120" -c "0120.sh"
mqsub -deps "2100"           -l "2001" -c "2001.sh"
mqsub -deps "2100"           -l "2200" -c "2200.sh"
mqsub -deps "1110 2001 2200" -l "1210" -c "1210.sh"
mqsub -deps "0120 1210"      -l "0220" -c "0220.sh"
mqsub -deps "1210"           -l "1111" -c "1111.sh"
mqsub -deps "0021 1111"      -l "0121" -c "0121.sh"
mqsub -deps "0121"           -l "0022" -c "0022.sh"
mqsub -deps "2100"           -l "3100" -c "3100.sh"
mqsub -deps "1110 3100"      -l "2110" -c "2110.sh"
mqsub -deps "0120 2110"      -l "1120" -c "1120.sh"
mqsub -deps "1120"           -l "0130" -c "0130.sh"
mqsub -deps "1110 2001"      -l "1011" -c "1011.sh"
mqsub -deps "0120 1011"      -l "0021" -c "0021.sh"
mqsub -deps "2001 3100"      -l "3001" -c "3001.sh"
mqsub -deps "1011 2110 3001" -l "2011" -c "2011.sh"
mqsub -deps "0021 1120 2011" -l "1021" -c "1021.sh"
mqsub -deps "0130 1021"      -l "0031" -c "0031.sh"
```

Figure 4.10: The flat submission script for the running example.

```
<target name>: <target 1> <target 2> ... <target n>
    <command line 1>
    <command line 2>
    ...
    <command line m>
```

Figure 4.11: The Makefile format.

4.2.1 Using Makefile

The standard UNIX utility `make` is regularly used to build software where constraints on the order of building of constituent components/modules must be imposed through the use of dependencies. Recall that a Makefile consists of *rules* that follow the format shown in Figure 4.11. A rule specifies how a single `target` is made. It starts with the name of the target to be made and the names of the targets that have to be made prior to the making of the target of interest. In Figure 4.11, the target `<target name>` can be made only after targets `<target 1>` through `<target n>` are made. Note that the order of making these targets is not specified and, in principle, they can be made concurrently. Finally, on separate lines indented by the tabulation character are the command lines that are executed to make the target.

The `make` utility itself does not impose any limitation on its usage. For example, instead of executing a compiler, one can execute a checkers computation. The execution of `make` on the Makefile in Figure 4.12 results in sequentially computing the workflow in Figure 4.7. It is possible to achieve concurrent execution of the Makefile in Figure 4.12 by using the so called *parallel* `make`, which is provided with many platforms and is invoked by `make -p`. However, the utilization of computational resources is limited to the physical machine on which the `make` utility is being executed.

To use TrellisDAG and take advantage of the available computational resources, we merely need to translate the commands of the Makefile in Figure 4.12 into invocations of `mqsub`. This is done by using the `mqtranslate` utility. We summarize this utility in Table 4.2.

The result of applying `mqtranslate` to the Makefile in Figure 4.12 is shown in Figure 4.13. The workflow can be submitted by simply applying `make` to the

```
all: 0022 0031

#compute 3-piece databases
2100:
	2100.sh
1110: 2100
	1110.sh
2001: 2100
	2001.sh
0120: 1110
	0120.sh
1011: 1110 2001
	1011.sh
0021: 0120 1011
	0021.sh

# 3 vs. 1
3100: 2100
	3100.sh
2110: 1110 3100
	2110.sh
3001: 2001 3100
	3001.sh
1120: 0120 2110
	1120.sh
2011: 1011 2110 3001
	2011.sh
0130: 1120
	0130.sh
1021: 0021 1120 2011
	1021.sh
0031: 0130 1021
	0031.sh

# 2 vs. 2
2200: 2100
	2200.sh
1210: 1110 2001 2200
	1210.sh
0220: 0120 1210
	0220.sh
1111: 1210
	1111.sh
0121: 0220 1111
	0121.sh
0022: 0121
	0022.sh
```

Figure 4.12: The Makefile for the running example.

Command/ Parameter	Description
mqtranslate	Reads a Makefile script from standard input and replaces all the commands with the invocations of `mqsub`. The resulting new Makefile is output to standard output. The parameters of `mqsub` are formed as follows: 1. The target of the command is used as an argument for `-l` 2. The dependencies of the target is used as an argument for `-deps` 3. The command is used as an argument for `-comm` 4. The comments immediately preceding the command and starting with `#attribute` (see Section 5.1) are interpreted as attributes and used as an argument for `-attr` 5. All other comments and lines are left without change

Table 4.2: Short description of the `mqtranslate` utility.

Makefile shown in Figure 4.13. Furthermore, applying `make -n` to that Makefile produces the flat submission script in Figure 4.10.

4.2.2 The DAG description script

Writing a flat submission script or a Makefile may be a cumbersome task, especially when the workflow contains hundreds or thousands of jobs as in the checkers computation. For some applications, it is possible to come up with simple naming conventions for the jobs and write a script to automatically produce a flat submission script or a Makefile. TrellisDAG helps the user by providing a framework for such a script. Moreover, through using this framework (which we call the *DAG description script*), the additional functionality of supergroups becomes available.

A DAG description script is a module coded using the Python scripting language; this module implements the interface required by TrellisDAG. TrellisDAG transforms that module into a Makefile and further into a flat submission script as described above.

We work our way up from using the most essential features of the DAG description script, to defining supergroups, to using job attributes.

```
all: 0022 0031

2100:
        mqsub -deps "" -l "2100" -c "2100.sh"
1110: 2100
        mqsub -deps "2100" -l "1110" -c "1110.sh"
2001: 2100
        mqsub -deps "2100" -l "2001" -c "2001.sh"
0120: 1110
        mqsub -deps "1110" -l "0120" -c "0120.sh"
1011: 1110 2001
        mqsub -deps "1110 2001" -l "1011" -c "1011.sh"
0021: 0120 1011
        mqsub -deps "0120 1011" -l "0021" -c "0021.sh"

3100: 2100
        mqsub -deps "2100" -l "3100" -c "3100.sh"
2110: 1110 3100
        mqsub -deps "1110 3100" -l "2110" -c "2110.sh"
3001: 2001 3100
        mqsub -deps "2001 3100" -l "3001" -c "3001.sh"
1120: 0120 2110
        mqsub -deps "0120 2110" -l "1120" -c "1120.sh"
2011: 1011 2110 3001
        mqsub -deps "1011 2110 3001" -l "2011" -c "2011.sh"
0130: 1120
        mqsub -deps "1120" -l "0130" -c "0130.sh"
1021: 0021 1120 2011
        mqsub -deps "0021 1120 2011" -l "1021" -c "1021.sh"
0031: 0130 1021
        mqsub -deps "0130 1021" -l "0031" -c "0031.sh"

2200: 2100
        mqsub -deps "2100" -l "2200" -c "2200.sh"
1210: 1110 2001 2200
        mqsub -deps "1110 2001 2200" -l "1210" -c "1210.sh"
0220: 0120 1210
        mqsub -deps "0120 1210" -l "0220" -c "0220.sh"
1111: 1210
        mqsub -deps "1210" -l "1111" -c "1111.sh"
0121: 0220 1111
        mqsub -deps "0220 1111" -l "0121" -c "0121.sh"
0022: 0121
        mqsub -deps "0121" -l "0022" -c "0022.sh"
```

Figure 4.13: The Makefile with calls to `mqsub` for the running example.

```
#!/usr/bin/python

import sys
import math
import string

nPieces = int(sys.argv[1])

def getDependentSlice(slice):
    return [ # dependencies come from retrograde analysis
        # a checker could have become a king
        [slice[0] + 1, slice[1], slice[2] - 1, slice[3]],
        [slice[0], slice[1] + 1, slice[2], slice[3] - 1],
        # a capture
        [slice[0] - 1, slice[1], slice[2], slice[3]],
        [slice[0], slice[1] - 1, slice[2], slice[3]],
        [slice[0], slice[1], slice[2] - 1, slice[3]],
        [slice[0], slice[1], slice[2], slice[3] - 1],
        # (switch black and white) + capture
        [slice[1] - 1, slice[0], slice[3], slice[2]],
        [slice[1], slice[0] - 1, slice[3], slice[2]],
        [slice[1], slice[0], slice[3] - 1, slice[2]],
        [slice[1], slice[0], slice[3], slice[2] - 1]]

#THE INTERFACE PART
Levels = [['Slice', 4]]

def generateGroup(level, superGroup):
    result = []
    #--- np - # of pieces
    #--- bk, wk - # of black and white kings
    #--- bc, wc - # of black and white checkers
    for np in range(3, nPieces + 1):
        for bk in range(0, np):
            for wk in range(0, np - bk + 1):
                for bc in range(0, np - bk - wk + 1):
                    # since np, bk, wk, bc are fixed, wc is computed
                    wc = np - bk - wk - bc
                    if wc < 0: break
                    if bk + bc == 0 or wk + wc == 0: continue
                    # elimination of slices based on symmetry
                    if bk + bc < wk + wc: continue
                    if bk + bc == wk + wc and bc < wc: continue
                    result.append([bk, wk, bc, wc])
    return result

def getPrologueExecutables(level, group): return []
def getPrologueAttributes(level, group): return []
def getPrologueParameters(level, group): return []
def getEpilogueExecutables(level, group): return []
def getEpilogueAttributes(level, group): return []
def getEpilogueParameters(level, group): return []

def getJobsExecutables(level, group):
    return [string.join(map(str, group), "") + ".sh"]

def getJobsAttributes(level, group): return [""]

def getJobsParameters(level, group): return [""]

def getDependent(level, group, superGroup):
    return getDependentSlice(group)
```

Figure 4.14: A basic DAG description script.

A basic DAG description script

The DAG description script shown in Figure 4.14 represents the workflow in Figure 4.7. Note that it reads one parameter from the command line (line 7): the maximal number of pieces on the board for the computation. The description script contains the following required declarations and definitions:

1. `Levels` – the list of used levels of groups (line 26). The levels are listed from highest (i.e. the level of groups that have no supergroups) to the lowest (i.e. the level of groups that have no subgroups). In our case, there is only one level that we call `Slice`. Each group at this level will be denoted by the name of the group appended by 4 numbers called the *vector-name* of the group. For example, the name of the group for slice `3100` is `Slice_3_1_0_0` and its vector name is `[3, 1, 0, 0]`,

2. `generateGroup` – the function for group generation (lines 28-45). The return value of this function is the list of groups (represented by their vector-names) at the given level with the given supergroup. In our case, there is only one level and we just return all slices to be computed (i.e. all of the slices corresponding to the nodes of the DAG in Figure 4.7),

3. `getDependent` – the function that specifies dependencies (lines 61 and 62); most of its functionality is implemented in the auxiliary function `getDependentSlice` (lines 9-23). The result of this function is a list of groups on which the given group depends. The system takes precautions to guarantee that all of the groups exist among those generated by the `generateGroup` function and have the same supergroup as the group of interest. That is why we just need to insure that the list that we return contains all the dependencies.

   For example, for the group with the vector name `[2, 0, 1, 1]`, the function returns the following vector names: `[3, 0, 0, 1]`, `[2, 1, 1, 0]`, `[1, 0, 1, 1]`, `[2, -1, 1, 1]`, `[2, 0, 0, 1]`, `[2, 0, 1, 0]`, `[-1, 2, 1, 1]`, `[0, 1, 1, 1]`, `[0, 2, 0, 1]`, `[0, 2, 1,`

`0]`. However, the system ignores all of the vector names except `[3, 0, 0, 1]`, `[2, 1, 1, 0]`, `[1, 0, 1, 1]`, `[2, 0, 0, 1]`, because the other vector names do not correspond to any of the groups generated by `generateGroup`.

Alternatively, the function `isDependent` can be provided. It takes two group parameters (second and third parameters) and returns 1 if the first group depends on the second group and 0 otherwise. Examples of description scripts that define this function can be found in Chapter 6,

4. Three functions are implemented to specify the jobs of a group:

   (a) `getJobsExecutables` must return the list of the names of executables for the individual jobs. In our case, each group has only one executable, whose name is closely related to the vector-name of the group. For example, the only job of the group with the vector name `[2, 1, 0, 0]` is `2100.sh`,

   (b) `getJobsParameters` must return the list of strings representing the command-line parameters passed to the jobs' executables. In our case, there are no command-line parameters to be passed and we match each job with an empty string, and

   (c) `getJobsAttributes` must return the list of strings representing the attributes of the jobs. We will concentrate on this feature in Section 4.2.4, and

5. The prologue jobs and epilogue jobs of a group are specified similarly as normal jobs. In our examples, we do not use this capability and the functions that are reserved for specifying prologue and epilogue jobs (`getPrologueExecutables`, `getPrologueParameters`, `getPrologueAttributes`, `getEpilogueExecutables`, `getEpilogueParameters` and `getEpilogueAttributes`) simply return an empty list.

Given a DAG description script, one can form a Makefile that can later be transformed into a flat submission script using the previously described utilities. The

Command/ Parameter	Description
mqmakemake	Calls functions of the DAG description script whose name is obtained by appending `.py` to the last command-line argument. The rest of the command line arguments are passed without change to the DAG description script. The result is the Makefile that is output to the standard output. The Makefile may contain special comments to encode levels and attributes (see Chapter 5 for more details).

Table 4.3: Short description of the `mqmakemake` utility.

Makefile is obtained by using the utility `mqmakemake`. If the DAG description script is called `running`, then the Makefile called `running.make` is obtained by issuing the command:

```
mqmakemake running > running.make
```

A short description of `mqmakemake` is given in Table 4.3.

Once the Makefile is obtained, we need to ensure that the defined levels are entered into the jobs database. This is done by means of calling the `mqmakelevels` utility as follows:

```
mqmakelevels running.make
```

More details about storing the information about levels in the jobs database is provided in Chapter 5.

4.2.3 The DAG description script with supergroups

In this section, we show how to modify the DAG description script in Figure 4.14 in order to obtain and submit the workflow shown in Figure 4.8. The new script is shown in Figure 4.15. We summarize the changes made to the script in Figure 4.14.

1. There are two levels (line 17). One of the levels corresponds to the supergroups in Figure 4.8 and is called `VS`. Note that 2 numbers naturally represent the groups at this level, one being the number of pieces of one color and the other being number of pieces of the other color.

2. There are two cases in the `generateGroup` function. In the case of `VS`

level (lines 22-24), we generate the groups `[[2, 1], [3, 1], [2, 2]]`. The generation of the groups at the level `Slice` for a given supergroup at the level `VS` is also straightforward (lines 26-34).

3. There are two cases in the function `getDependent`. The implementation for the case of `VS` level is given in function `getDependentVS`. The implementation for the case of `Slice` level is simpler than it was in the description script without supergroups, because we only need to worry about dependencies within one group at `VS` level (lines 9-11).

4. Finally, the groups at level `VS` do not have jobs and the functions `getJobsExecutables`, `getJobsParameters` and `getJobsAttributes` are modified accordingly.

```
#!/usr/bin/python

import sys
import math
import string

nPieces = int(sys.argv[1])

def getDependentSlice(slice):
    return [[slice[0] + 1, slice[1], slice[2] - 1, slice[3]],
            [slice[0], slice[1] + 1, slice[2], slice[3] - 1]]

def getDependentVS(vs):
    return [[vs[0] - 1, vs[1]], [vs[0], vs[1] - 1]]

#THE INTERFACE PART
Levels = [['VS', 2], ['Slice', 4]]

def generateGroup(level, parentGroup):
    result = []
    if level == 'VS':
        for np in range(3, nPieces + 1):
            for b in range((np + 1)/2, np):
                result.append([b, np - b])
    else:
        b = parentGroup[0]
        w = parentGroup[1]
        for bk in range(0, b + 1):
            for wk in range(0, w + 1):
                bc = b - bk
                wc = w - wk
                if bk + bc < wk + wc: continue
                if bk + bc == wk + wc and bc < wc: continue
                result.append([bk, wk, bc, wc])
    return result

def getPrologueExecutables(level, group): return []
def getPrologueAttributes(level, group): return []
def getPrologueParameters(level, group): return []
def getEpilogueExecutables(level, group): return []
def getEpilogueAttributes(level, group): return []
def getEpilogueParameters(level, group): return []

def getJobsExecutables(level, group):
    if level != 'Slice': return []
    return [string.join(map(str, group), "") + ".sh"]

def getJobsAttributes(level, group):
    if level != 'Slice': return []
    return [""]

def getJobsParameters(level, group):
    if level != 'Slice': return []
    return [""]

def getDependent(level, group, parentGroup):
    if level == 'VS':
        return getDependentVS(group)
    else:
        return getDependentSlice(group)
```

Figure 4.15: The DAG description script with supergroups.

4.2.4 The DAG description script with attributes

Sometimes it is convenient to associate more information with a job than just the command line. Such information can be used by the scheduling system or by the user. In TrellisDAG, the extra information associated with individual jobs is stored as key-value pairs called *attributes*. In this version of the system, we have several kinds of attributes that serve the following purposes:

1. Increasing the degree of concurrency by relaxing workflow dependencies,
2. Regulating the placement of jobs, i.e. mapping jobs to execution hosts, and
3. Storing the history profile – when and where jobs were started and completed (also see the note to Table 4.4).

In this section, we concentrate on the first two kinds of attributes.

We note that a higher degree of concurrency can be achieved if we separate computation of the checkers end-game databases with their verification. For example, we can split the script for computing the `0121` slice into 2 jobs: `0121.comp.sh` and `0121.ver.sh`. Once the former job is done, we can start the computation for the `0022` slice (i.e. start `0022.comp.sh`).

We would like to reuse the script shown in Figure 4.15 as much as possible. So, we keep the computation and the verification as jobs of one group at the `Slice` level. However, we introduce an attribute that allows the dependent groups to proceed when a group reaches a certain point in the computation (i.e. a certain number of jobs are complete). We refer to this feature as *early release*. In our example, the computation job of all slices will have the `release` attribute set to `yes` and that will result in the workflow dependency of the dependent groups being satisfied once the computation is complete.

We also take into account that verification needs the data that has been produced during the computation. Therefore, it is desirable that verification runs on the same machine as the computation. Hence, we introduce another attribute called *affinity*. When the `affinity` of a job is set to `yes`, the job is forced to be executed on the same machine as the previous job of its group.

In the DAG description script in Figure 4.15, we only had to change the functions `getJobsExecutables`, `getJobsParameters` and `getJobsAttributes`. The modified description script is shown in Figure 4.16.

```
#!/usr/bin/python

import sys
import math
import string

nPieces = int(sys.argv[1])

def getDependentSlice(slice):
    return [[slice[0] + 1, slice[1], slice[2] - 1, slice[3]],
            [slice[0], slice[1] + 1, slice[2], slice[3] - 1]]

def getDependentVS(vs):
    return [[vs[0] - 1, vs[1]], [vs[0], vs[1] - 1]]

#THE INTERFACE PART
Levels = [['VS', 2], ['Slice', 4]]

def generateGroup(level, parentGroup):
    result = []
    if level == 'VS':
        for np in range(3, nPieces + 1):
            for b in range((np + 1)/2, np):
                result.append([b, np - b])
    else:
        b = parentGroup[0]
        w = parentGroup[1]
        for bk in range(0, b + 1):
            for wk in range(0, w + 1):
                bc = b - bk
                wc = w - wk
                if bk + bc < wk + wc: continue
                if bk + bc == wk + wc and bc < wc: continue
                result.append([bk, wk, bc, wc])
    return result

def getPrologueExecutables(level, group): return []
def getPrologueAttributes(level, group): return []
def getPrologueParameters(level, group): return []
def getEpilogueExecutables(level, group): return []
def getEpilogueAttributes(level, group): return []
def getEpilogueParameters(level, group): return []

def getJobsExecutables(level, group):
    if level != 'Slice': return []
    return [string.join(map(str, group), "") + ".comp.sh",
            string.join(map(str, group), "") + ".ver.sh"]

def getJobsAttributes(level, group):
    if level != 'Slice': return []
    return [["affinity=no", "release=yes"],
            ["affinity=yes", "release=no"]]

def getJobsParameters(level, group):
    if level != 'Slice': return []
    return ["", ""]

def getDependent(level, group, parentGroup):
    if level == 'VS':
        return getDependentVS(group)
    else:
        return getDependentSlice(group)
```

Figure 4.16: The DAG description script with supergroups and attributes.

Service	Description
mqnextjob	Obtain ID of a ready job
mqgetjobcommand	Get command line given a job ID
mqgetjobattributes	Get a value of a specific attribute of a job with a given ID
mqstatus	Inform the system of the status of computation of a job
mqsignal	Inform the system that the placeholder is alive
mqlock	Obtain a named exclusive lock
mqunlock	Release a lock with the given name
mqdonejob	Inform the system that the job with the given ID is complete

Table 4.4: Services of the command-line server. The order of presentation is the order in which the services are usually invoked in the placeholder. Note: `mqnextjob` and `mqdonejob` modify the `lastStart`, `lastCompletion` and `lastHost` attributes of the job in order to store when (and on which host) the job was started and completed in the last run of the computation.

4.3 Services of the command-line server

Services of the command-line server (see Figure 4.1) are the programs through which a placeholder can access and modify the jobs database. These services are normally called within an `ssh` session in the placeholder. All of the services are listed and briefly described in Table 4.4.

All of the services get their parameters on the command line as key-value pairs. For example, if the value associated with the key `id` is 5, then the key-value pair on the command line is `id=5`.

We start with the service called `mqnextjob`. The output of this service represents the job ID of the job that is scheduled to be run by the calling placeholder. The keys for this service are summarized in Table 4.5. For example, the service could be called as follows:

```
ssh server ‘‘mqnextjob sched=SGE \
sched_id=$JOB_ID \
submit_host=brule host=‘hostname‘’’
```

Once the job's ID is obtained, we can obtain the command line for that job using the `mqgetjobcommand` service. The command line is output to the standard output. The keys for this service are summarized in Table 4.6.

Key	Description
sched	The name of the local scheduler, e.g. SGE.
submit_host	The submission host for the local scheduler
sched_id	The ID given to the placeholder by the local scheduler, e.g. $JOBID in SGE
host	The name of the host on which the placeholder is running

Table 4.5: Keys for the `mqnextjob` service.

Key	Description
id	The job ID of the job

Table 4.6: Keys for the `mqgetjobcommand` service.

The placeholder has access to the attributes of a job through the `mqgetjobattribute` service. The service outputs the value associated with the given attribute. The keys for this service are summarized in Table 4.7.

When the job is complete, the placeholder has to inform the system about this event. This is done using the `mqdonejob` service. The keys for this service are summarized in Table 4.8.

The user can maintain his own status of the running job in the jobs database by using the `mqstatus` service. The status entered through this service is shown by the `mqstat` monitoring utility (see Section 4.4.1). The keys for this service are summarized in Table 4.9.

The `mqsignal` service is the means by which the placeholder can assure the system that it is still alive and functioning normally. For example, the placeholder can spawn a background script that would call the service at equal time intervals. We will show an example of such a script in Section 6.5. A script can be running on the server that checks when the service was invoked last time and takes actions when

Key	Description
id	The job ID of the job
attribute	The name of the attribute

Table 4.7: Keys for the `mqgetjobattributes` service.

Key	Description
id	The job ID of the job

Table 4.8: Keys for the `mqdonejob` service.

Key	Description
id	The job ID of the job
status	The string representing the job's or placeholder's status

Table 4.9: Keys for the `mqstatus` service.

the gap between two consecutive calls to `mqsignal` is too large. We implemented such a script and called it `mqdaemon`, shown in Appendix F. The keys for this service are summarized in Table 4.10.

Finally, there are two services that support named exclusive locks. Using these locks, for example, it is possible to implement critical sections within placeholders. The `mqlock` service is used to obtain the lock and assign a name to it and the `mqunlock` service is used to release a lock with the given name. The keys for these services are summarized in Tables 4.11 and 4.12. In this version, locks are associated with jobs rather than placeholders and thus can only be used after `mqnextjob` has been invoked.

Key	Description
id	The job ID of the job

Table 4.10: Keys for the `mqsignal` service.

Key	Description
id	The job ID of the job
name	The name of the lock
host	Name of the host that will hold the lock

Table 4.11: Keys for the `mqlock` service.

Key	Description
name	The name of the lock

Table 4.12: Keys for the `mqunlock` service.

4.4 Utilities

The utilities described in this section are used to monitor the computation and perform some basic administration tasks.

4.4.1 `mqstat`: status of the computation

This utility outputs information about running jobs: job ID, command line, hostname of the machine where the job is running, time when the job has started, and the status submitted by means of calling the `mqstatus` service.

4.4.2 `mqdump`: jobs browser

`mqdump` is an interactive tool for browsing, monitoring, and changing the dynamic status of groups. The first screen of `mqdump` for the example corresponding to the DAG description script in Figure 4.16 is presented in Figure 4.17. Following the prompt, the user types the number from 1 to 11 corresponding to the command that must be executed. If the selected command operates on one or more groups, then the desired groups' numbers (see the left column on top of Figure 4.17) are typed as a comma-separated list in response to the next prompt of `mqdump`.

We see that groups are numbered from 0 to 5 and each group has a unique identification number. Each group has two statuses: one shows the progress of the computation associated with the group and its subgroups, while the other shows the progress with respect to the attributes of the group's jobs. Currently, the only attribute statuses used are `None` and `Released`.

There are 11 commands and we explain each of them. Note that some commands are applied to specific groups. For most such commands, the user can specify a comma-separated list of groups and the command will be applied to the maximal possible set of these groups. The system makes sure that no action can result in an inconsistent (with respect to dependencies) state of the jobs database.

1. `Dive` – the selected group's subgroups are shown and the level changes to the levels of the subgroups. Only one group can be selected for this command.

```
Command  ID         Group                          Status                   Attribute Status
-------- ---------- ------------------------------ ------------------------ ---------------
0.       1          VS_2_1_prologue                Not started              None
1.       14         VS_2_1                         Not started              None
2.       15         VS_2_2_prologue                Not started              None
3.       28         VS_2_2                         Not started              None
4.       29         VS_3_1_prologue                Not started              None
5.       46         VS_3_1                         Not started              None

1. Dive
2. Show dependencies
3. Enable        4. Disable
5. Not started   6. Done
7. Stop          8. Stop-Disable
9. Up
10. Refresh
11. Quit
Command?(1-11)
```

Figure 4.17: The first screen of `mqdump` for the running example.

2. `Show dependencies` – only the groups of the current level that have to be executed before the selected group are shown. Only one group can be selected for this command.

3. `Enable` – enable the selected groups. The command takes action only on groups that were previously disabled.

4. `Disable` – disable the selected groups. The groups on which this command succeeded and their subgroups cannot be started.

5. `Not started` – force the selected groups to be recomputed.

6. `Done` – inform the system that the computation for the selected groups have been performed (e.g. by hand).

7. `Stop` – not implemented.

8. `Stop-Disable` – not implemented.

9. `Up` – go one level higher. The level is changed to that of the supergroup of the groups at the current level.

10. `Refresh` – To speed up operation of the utility, we chose to make a snapshot of the jobs database at the beginning and work with that snapshot unless

the user changes the statuses of some groups by means of the commands described above. The `Refresh` command forces the utility to make a new snapshot of the jobs database. The current level is not changed.

11. `Quit` – exit the program.

`mqenable`, `mqdisable.py`, `mqnotstarted`, **and** `mqdone`

These are batch mode facilities that correspond to the commands of `mqdump`. Each of these utilities take a list of unique group IDs (as shown by the `mqdump` utility) or names on the command line and performs the required action.

4.5 Concluding remarks

The features of TrellisDAG can be summarized as follows (see Figure 4.1):

1. The workflow and its dependencies can be described in several ways (see Figure 4.9). The user chooses the way of description based on the properties of the workflow (such as the level of complexity) and his own preferences,

2. Whatever way of description the user chooses, utilities are provided to translate this description into the jobs database,

3. Services of the command-line server are provided so that the user can access and modify the jobs database dynamically, either from the command line or from within a placeholder,

4. `mqstat` and `mqdump` are utilities that can be used to perform monitoring and simple administrative tasks, and

5. The history profile of the last computation is stored in the jobs' attributes. A user possessing programming skills can implement a scheduling policy that best suits his specific application (see Section 5.3.1 for details).

In the next chapter, we provide a high-level description of the implementation of the features of TrellisDAG that were presented in this chapter.

Chapter 5

Implementation of TrellisDAG

In Chapter 4, we described our system at a high level. In this chapter, we provide some details about the implementation of the features that were presented in Chapter 4.

We start by explaining the Makefile that TrellisDAG produces from the DAG description script. In Section 5.2, we describe the schema of the jobs database. Section 5.3 gives pseudo code for several services of the command-line server. In Section 5.4, we explain how the services of the `mqdump` utility can be implemented using the jobs database schema described in Section 5.2.

5.1 `mqmakemake`: generation of the Makefile

Jobs of a group can only start after three prerequisites have been satisfied:

1. The workflow dependencies are satisfied, i.e. the groups with the same supergroup on which the group explicitly depends are complete,

2. The subgroups of the group are complete, and

3. The prologue jobs of the group are complete (see Figure 5.1).

Generating a Makefile that would preserve dependencies between jobs as well as allow for prologue and epilogue jobs turned out to be a non-trivial problem. We illustrate our solution to this problem with an example. Figures 5.2 and 5.3 show a part of the Makefile that was generated by `mqmakemake` for the DAG description script in Figure 4.16. This part includes all of the targets for the 3-piece databases.

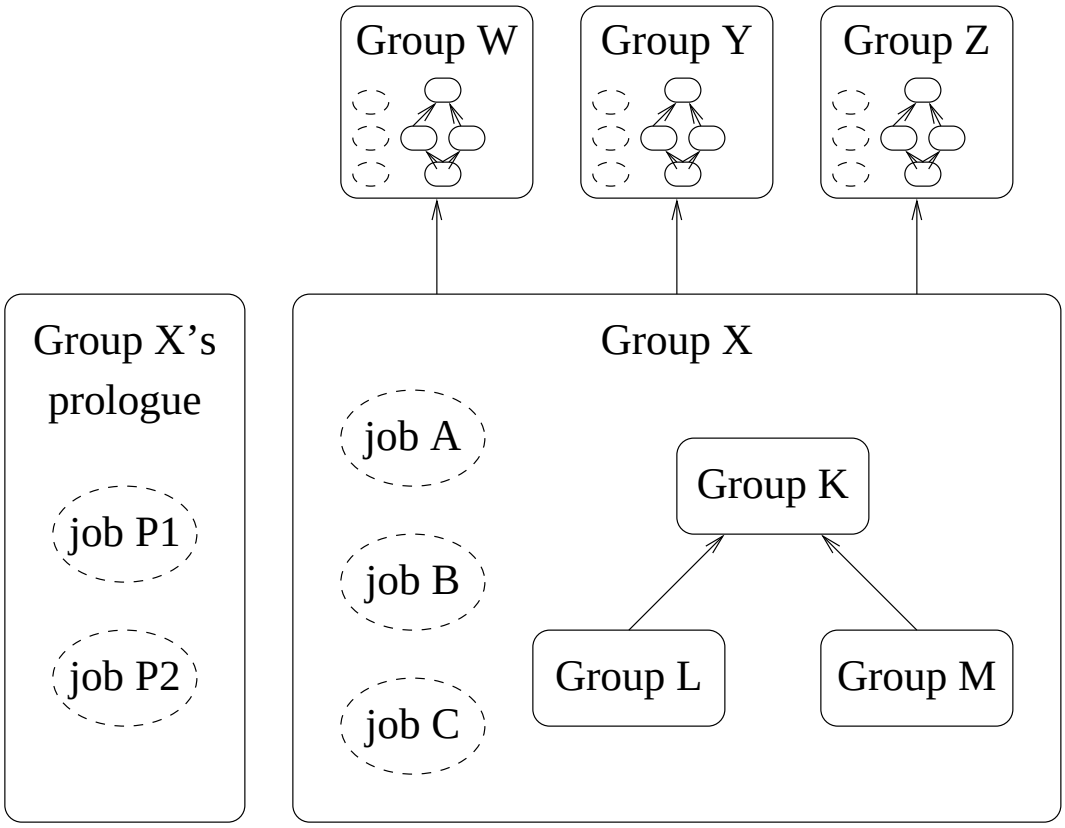

Figure 5.1: When the jobs of groups W, Y, Z are complete, the jobs of group X's prologue can be executed. Then the jobs of groups K, L, M are executed. Only after that does job A of group X become ready.

We start by deciding that each group will correspond to a single target/rule of the Makefile (the Makefile format was discussed in Section 4.2.1). The commands of a rule are the command lines of the jobs of the corresponding group. Whenever a target does not have any commands, we give it an `echo` command. For example, the target `VS_2_1` corresponds to the group of positions with 2 pieces of one color and 1 piece of the other color. Although all of the jobs for this group are contained in its subgroups, we need a representation of the group in the jobs database and the auxiliary `echo` job serves this purpose.

Note that we cannot just list all of the prerequisites of the jobs of a group in the dependencies part of the corresponding rule, because all of the dependencies can be built simultaneously (refer to Section 4.2.1).

```
 1  #AutoMakefile
 2  #VS
 3  #Slice
 4
 5  all:  VS_2_1  VS_2_2  VS_3_1
 6
 7  VS_2_1:  Slice_0_0_2_1  Slice_0_1_2_0  Slice_1_0_1_1  Slice_1_1_1_0 \
           Slice_2_0_0_1  Slice_2_1_0_0
 8          echo
 9
10  VS_2_1_prologue:
11          echo
12
13  Slice_0_0_2_1:  Slice_0_0_2_1_prologue
14  #attribute affinity=no
15  #attribute release=yes
16          0021comp.sh
17  #attribute affinity=yes
18  #attribute release=no
19          0021ver.sh
20
21  Slice_0_0_2_1_prologue:  Slice_1_0_1_1 Slice_0_1_2_0
22          echo
23
24  Slice_0_1_2_0:  Slice_0_1_2_0_prologue
25  #attribute affinity=no
26  #attribute release=yes
27          0120comp.sh
28  #attribute affinity=yes
29  #attribute release=no
30          0120ver.sh
31
32  Slice_0_1_2_0_prologue:  Slice_1_1_1_0
33          echo
34
35  Slice_1_0_1_1:  Slice_1_0_1_1_prologue
36  #attribute affinity=no
37  #attribute release=yes
38          1011comp.sh
39  #attribute affinity=yes
40  #attribute release=no
41          1011ver.sh
42
```

Figure 5.2: The Makefile produced by `mqmakemake` (continued in Figure 5.3).

Therefore, we need a more sophisticated organization of the Makefile. The structure of the Makefile that we came up with is summarized as follows:

1. The target of a group at the lowest level depends on the target of its prologue group,

2. The target of a group that has subgroups depends on their targets,

3. The target of the prologue group of a group that has workflow dependencies, depends on the targets of the parent groups of that group (i.e. it depends on the groups corresponding to the workflow dependencies), and

```
43  Slice_1_0_1_1_prologue:  Slice_2_0_0_1 Slice_1_1_1_0
44          echo
45
46  Slice_1_1_1_0:  Slice_1_1_1_0_prologue
47  #attribute affinity=no
48  #attribute release=yes
49          1110comp.sh
50  #attribute affinity=yes
51  #attribute release=no
52          1110ver.sh
53
54  Slice_1_1_1_0_prologue:  Slice_2_1_0_0
55          echo
56
57  Slice_2_0_0_1:  Slice_2_0_0_1_prologue
58  #attribute affinity=no
59  #attribute release=yes
60          2001comp.sh
61  #attribute affinity=yes
62  #attribute release=no
63          2001ver.sh
64
65  Slice_2_0_0_1_prologue:  Slice_2_1_0_0
66          echo
67
68  Slice_2_1_0_0:  Slice_2_1_0_0_prologue
69  #attribute affinity=no
70  #attribute release=yes
71          2100comp.sh
72  #attribute affinity=yes
73  #attribute release=no
74          2100ver.sh
75
76  Slice_2_1_0_0_prologue:  VS_2_1_prologue
77          echo
```

Figure 5.3: The Makefile produced by `mqmakemake` (beginning in Figure 5.2).

4. The target of the prologue group of a group that does not have workflow dependencies, depends on the target of the prologue group of the supergroup of that group.

We illustrate these rules using the Makefile in Figures 5.2 and 5.3.

1. The group `2100` at the `Slice` level is an example of a group at the lowest level. Therefore, the target corresponding to this group `Slice_2_1_0_0` depends on the target corresponding to its prologue group `Slice_2_1_0_0_prologue` (line 68).

2. The supergroup of `2100` is the group of all slices with 2 pieces of one color and 1 piece of the other color. The target corresponding to this group `VS_2_1` depends on all of the targets corresponding to its subgroups (line

7): `Slice_0_0_2_1`, `Slice_0_1_2_0`, `Slice_1_0_1_1`, `Slice_1_1_1_0`, `Slice_2_0_0_1`, `Slice_2_1_0_0`.

3. `0021` is an example of a group that has workflow dependencies. Its dependencies are `1011` and `0120`. That is why the target corresponding to the prologue group `Slice_0_0_2_1_prologue` depends on the targets corresponding to those dependencies (line 21): `Slice_1_0_1_1` and `Slice_0_1_2_0`.

4. `2100` is an example of a group that does not have workflow dependencies. That is why the target of its prologue group `Slice_2_1_0_0_prologue` depends on the target corresponding to the prologue of its supergroup `VS_2_1_prologue` (line 76).

5.2 The jobs database

We use the PostgreSQL relational database management system [35] to store the jobs of the computation. The SQL script for creating the tables of the jobs database is shown in Figure 5.4. We need to store more information than just the command lines associated with the jobs to be executed. For example,

1. Information to identify group and super-/subgroup relationships (table `Levels` and attribute `level_id` in table `Targets` in Figure 5.4),

2. Information about workflow dependencies between groups (table `Before` in Figure 5.4), and

3. Extra information needed for the operation of `mqstat` and `mqdump` utilities has to be stored (there are several status and time fields in `Running` and other tables in Figure 5.4).

Examples of instantiated tables are given in Figures 5.5, 5.6, 5.7, 5.8, 5.9 and 5.10, where we show the part of the jobs database for the DAG description script in Figure 4.16 that corresponds to the piece of the Makefile shown in Figures 5.2 and 5.3.

The `Levels` table is shown in Figure 5.5. It simply stores the names of the levels with corresponding ids.

The `Targets` table is shown in Figure 5.6. Each tuple in the table corresponds to a target in the `Makefile` shown in Figures 5.2 and 5.3. Note that each target cross-references a unique level. Although the information about groups and super- and sub-relationships is not stored in the database explicitly, the `level_id` field of the `Targets` table makes it possible to identify groups (discussed in Section 5.4).

The `Jobs` table is shown in Figure 5.7. The main responsibility of the `Jobs` table is to store the command lines of the jobs. We note that there are quite a few jobs with the simple command line `echo`. This was initially done for simplicity of implementation, so that each target in the `Targets` table corresponds to at least one job in the `Jobs` table. The extra jobs do not affect performance much (they are never given to placeholders for execution) and still remain in this version. Effectively, the `echo`'s are no-operation jobs (no-ops).

The `Before` table is shown in Figure 5.8. The table keeps the dependencies as presented by the `-deps` parameter of `mqsub` commands (which corresponds in one-to-one fashion to the dependencies in the `Makefile`). The value in the `status` field shows whether the dependency has been dynamically satisfied or not. This is the only dynamic field in the `Before` table.

The `Running` table represents the jobs that have been given out to the placeholders for execution by the `mqnextjob` service. Thus all fields of the `Running` table are dynamic. Immediately after the flat submission script has run, the table is empty since no jobs have been assigned to placeholders. Let us invoke `mqnextjob` and see how the table will be filled. Here we show that `mqnextjob` can be invoked from the command line by the user. Usually, however, it is invoked from a placeholder as shown in Section 4.3.

```
mqnextjob sched=sge sched_id=1 \
submit_host=server host=machineA
```

The `Running` table after execution of the above command is shown in Figure 5.9. Many of the fields repeat (for convenience of implementation) the informa-

tion about the job with ID 3 from the `Jobs` table (including `status`, which has changed to 3 in the `Jobs` table as well). Other pieces of information include:

1. `started` – the system time (measured in seconds from 00:00 of January 1, 1970) when the job was given out for execution,

2. `user_status` and `user_started` – the status entered by means of the `mqstatus` service and the system time when that status was last changed. As we see the time stamp is initialized to the value in the `started` field, and

3. `last_signal` – the system time when the `mqsignal` service was last invoked.

Since jobs of a group execute in the pipeline order, we know that whenever a job completes, the next job from the same group is ready for execution. Thus we can often reduce the time required for the execution of the `mqnextjob` service by storing the information about the next job of the group in the `mqdonejob` service. Let us invoke `mqdonejob` on the job with ID 3 and see how the described information is saved in the `Cache` table.

```
mqdonejob id=3
```

The `Cache` table after execution of the above command is shown in Figure 5.10. The only tuple in the table says that the job with ID 4 is ready to be executed and that this job had the `affinity` attribute set to `yes` and therefore must be executed on the same machine on which the previous job of the same group was executed, namely on `machineA`. Note that all fields of the `Cache` table are dynamic.

```
 1  CREATE TABLE Levels (
 2          level_id int PRIMARY KEY,
 3          level_name      varchar(64) UNIQUE
 4  );
 5
 6  CREATE TABLE Targets (
 7          tar_id  int PRIMARY KEY,
 8          tar_name        varchar(64) UNIQUE,
 9          level_id        int REFERENCES Levels,
10          status  int,                    -- DYNAMIC!
11          attr_status int                 -- DYNAMIC!
12  );
13
14  CREATE TABLE Jobs (
15          j_id int PRIMARY KEY,           -- unique job id
16          tar_id  int REFERENCES Targets, -- target id
17          comm_line       varchar(800),   -- command line
18          status int,                     -- DYNAMIC!
19          attrs   varchar(100)            -- SOME ATTRS ARE DYNAMIC!
20  );
21
22  CREATE TABLE  Before (
23          pre_id  int REFERENCES Targets,
24          dep_id  int REFERENCES Targets,
25          status int,                     -- DYNAMIC!
26          PRIMARY KEY (pre_id, dep_id)
27  );
28
29  CREATE TABLE Running (                  -- ALL FIELDS ARE DYNAMIC!
30          j_id int UNIQUE REFERENCES Jobs,
31          comm_line       varchar(256),   -- command line
32          hostname        varchar(20),
33          started float,                  -- time when started
34          status int,
35          attrs   varchar(100),
36          user_status        varchar(40),
37          user_started float,             -- time when user_status
38                                          -- is last changed
39          scheduler        varchar(5),    -- the local scheduler
40          sched_id        varchar(10),    -- the id given to the job
41                                          -- by local scheduler
42          submit_host        varchar(20),
43          last_signal     float,          -- time when the job last
44                                          -- signaled that it's alive
45          PRIMARY KEY (j_id)
46  );
47
48  CREATE TABLE Cache (                    -- ALL FIELDS ARE DYNAMIC!
49          j_id    int UNIQUE REFERENCES Jobs,
50          hostname        varchar(20),    -- the required host
51          PRIMARY KEY (j_id)
52  );
53
54  CREATE TABLE Locks (                    -- ALL FIELDS ARE DYNAMIC!
55          lock_id int PRIMARY KEY,
56          lock_name       varchar(64) UNIQUE,
57          host    varchar(64),
58          started float,                  -- time when the lock was obtained
59          j_id    int REFERENCES Jobs
60  );
```

Figure 5.4: SQL script for creating the tables of the jobs database.

level_id	level_name
0	VS
1	Slice

Figure 5.5: The `Levels` table.

tar_id	tar_name	level_id	status	attr_status
1	VS_2_1_prologue	0	0	0
2	Slice_2_1_0_0_prologue	1	0	0
3	Slice_2_1_0_0	1	0	0
4	Slice_2_0_0_1_prologue	1	0	0
5	Slice_2_0_0_1	1	0	0
6	Slice_1_1_1_0_prologue	1	0	0
7	Slice_1_1_1_0	1	0	0
8	Slice_1_0_1_1_prologue	1	0	0
9	Slice_1_0_1_1	1	0	0
10	Slice_0_1_2_0_prologue	1	0	0
11	Slice_0_1_2_0	1	0	0
12	Slice_0_0_2_1_prologue	1	0	0
13	Slice_0_0_2_1	1	0	0
14	VS_2_1	0	0	0

Figure 5.6: The `Targets` table.

j_id	tar_id	comm_line	status	attrs
1	1	echo	0	
2	2	echo	0	
3	3	2100comp.sh	0	affinity=no#release=yes
4	3	2100ver.sh	0	affinity=yes#release=no
5	4	echo	0	
6	5	2001comp.sh	0	affinity=no#release=yes
7	5	2001ver.sh	0	affinity=yes#release=no
8	6	echo	0	
9	7	1110comp.sh	0	affinity=no#release=yes
10	7	1110ver.sh	0	affinity=yes#release=no
11	8	echo	0	
12	9	1011comp.sh	0	affinity=no#release=yes
13	9	1011ver.sh	0	affinity=yes#release=no
14	10	echo	0	
15	11	0120comp.sh	0	affinity=no#release=yes
16	11	0120ver.sh	0	affinity=yes#release=no
17	12	echo	0	
18	13	0021comp.sh	0	affinity=no#release=yes
19	13	0021ver.sh	0	affinity=yes#release=no
20	14	echo	0	

Figure 5.7: The `Jobs` table.

pre_id	dep_id	status
1	2	0
2	3	0
3	4	0
4	5	0
3	6	0
6	7	0
5	8	0
7	8	0
8	9	0
7	10	0
10	11	0
9	12	0
11	12	0
12	13	0
13	14	0
11	14	0
9	14	0
7	14	0
5	14	0
3	14	0

Figure 5.8: The `Before` table.

j_id	comm_line	hostname	started	status	attrs
3	2100comp.sh	machineA	1045710325.46	3	affinity=no#release=yes

user_status	user_started	scheduler	sched_id	submit_host	last_signal
	1045710325.46	sge	1	server	1045710325.46

Figure 5.9: The `Running` table.

j_id	hostname
4	machineA

Figure 5.10: The `Cache` table.

5.3 Services of the command-line server

Services of the command-line server (see Figure 4.1) are the programs that let the user (or a placeholder) dynamically access and modify the jobs database. In this respect, the most important operations are the ones that let us get a ready job's ID and notify the system upon completion of a job. These functions are made available through the `mqnextjob` and `mqdonejob` services of the command-line server (see Section 4.3). In this section, we give the pseudo-code for these services. The complete code listing is provided in Appendix D.

5.3.1 Implementation of `mqnextjob`

The workings of this service can be described by the following steps of an algorithm:

1. If the placeholder (which is uniquely identified by the values of `sched`, `sched_id` and `submit_host`) that is requesting a job is already running a job, then output the ID of that job and exit.

2. If there are no tuples in the `Cache` table or, if all tuples in the `Cache` table correspond to running jobs, then look for targets whose dependencies have been satisfied (this is done by querying the `Before` table) and put the first ready jobs of these targets into the `Cache` table; otherwise, go to the next step. If no tuples could be added to the `Cache` table, then go to the last step.

3. While there are tuples in the `Cache` table that correspond to the jobs with `echo` command lines, mark these jobs as done (see Section 5.3.2) and put the jobs that became ready due to this change into the table.

4. Select a non-running job in the `Cache` table, which has to be executed on the same host as the placeholder requesting a job. If there is no such job, then select a job for which the name of the host is not specified. In this version of the system, whenever a choice is not unique we rely on PostgreSQL to select a job with minimal job ID.

5. If a job is selected, then do the following:

   (a) Output the job's ID as a result.

   (b) Change the status information in the `Targets` table (this may involve changes to the `status` fields of the targets corresponding to the supergroups up to the highest level).

   (c) Change the `status` field in the `Jobs` table.

   (d) Insert a tuple into the `Running` table.

   (e) Change the `lastStart` attribute of the job to be the current system time.

   (f) Exit.

6. If there are still non-completed jobs, output 0 and exit. Otherwise, output -1 and exit.

The first step of the algorithm is a simple fault-tolerance measure that ensures that the `mqnextjob` service is idempotent with respect to two calls in a row (i.e. without a call to `mqdonejob` in between) to this service from the same placeholder. It can happen that `mqnextjob` is executed in an `ssh` session that gets broken after `mqnextjob` has succeeded. If the placeholder implements a retry mechanism, it will try to obtain a job again, in which case this step of the algorithm will be invoked.

The precondition of Step #2 is normally true only once at the beginning of the computation. After that, the `Cache` table contains at least one ready job.

Step #4 is the implementation of a simple scheduling policy. Other policies can be implemented (see Section 7.1).

5.3.2 Implementation of `mqdonejob`

`mqdonejob` does the following:

1. If a job with the given ID is not found in the `Running` table, then exit.

2. Remove the corresponding tuple in the `Cache` and `Running` tables and update the `status` field in the `Jobs` table.

3. If this was the last job of its target or if the `release` attribute is set to `yes`, then update the `status` field in the `Before` table. In the former case, also update the `status` field in the `Targets` table.

4. Put into the `Cache` table the tuples corresponding to the jobs that became available due to completion of the job. If the completed job was not the last job in its target and its `release` attribute was not set to `yes`, then the only job that could become available is the next job in the target; if that job has the `affinity` attribute set to `yes`, then we also fill in the `host` field with the name of the host where the completed job was run.

5. Set `lastCompletion` and `lastHost` attributes of the completed job to the current system time and the host name on which the job was run, respectively.

5.4 Services of `mqdump`

So far in this chapter we have been discussing the implementation of the features that make the automatic submission and computation of a workflow possible. In Section 4.4, we introduced utilities that the user can execute in order to conveniently monitor the computation and perform some administrative tasks. In this section, we discuss how a monitoring utility was implemented based on the jobs database's schema introduced in Section 5.2.

The only difficulty that we need to overcome here is that the information about groups is not stored explicitly in the jobs database. Instead, we have a level number associated with each target (level 0 corresponds to the highest level, level 1 to the second highest, etc.).

All operations can be expressed in terms of these two:

1. **Supergroup operator**: Given the target corresponding to a group, find the target corresponding to its supergroup and

2. **Subgroups operator**: Given the target corresponding a group, find the targets corresponding to its subgroups.

For example, to find all groups that have the same supergroup as the given group, we perform the above two operations in order. Although in practice many operations do not require the use of the two operations above, by showing how these two operations can be implemented, we show that the information stored in the jobs database is sufficient to retrieve all information about groups and super-/subgroup relationships.

The algorithm for the supergroup operator is as follows:

> Starting with the given target, perform a breadth-first search of targets, where the children of a target are the targets that depend on it. Stop when a target with a higher level than that of the given target is encountered. That target is the target corresponding to the supergroup.

The algorithm for the subgroups operator is as follows:

1. Starting with the given target, perform a breadth-first search of targets, where the children of a target are the targets on which it depends. Do not expand any children at the same or higher level as the initial target. Store the expanded targets in a list.

2. From the list formed in the previous step, select those targets whose level number is greater by 1 than the level number of the given target. These targets correspond to the subgroups of the group corresponding to the given target.

5.5 Concluding remarks

We presented a high-level description of our implementation of TrellisDAG. The use of the technique of placeholder scheduling made for several easy design choices (see Figure 2.2):

1. Service routines of the command-line server have to access and modify the storage of the command lines and

2. Since multiple placeholders can invoke the services of the command-line server at the same time, we can make use of the transaction-oriented properties of relational database management systems. We decided to use PostgreSQL because it was free and readily available.

For the same reason (i.e. using placeholder scheduling), the implementation of TrellisDAG is quite compact.

Chapter 6

Experimental Assessment of TrellisDAG

In this chapter, we present three experiments that show how TrellisDAG can be applied to compute different parts of the checkers endgame databases. The checkers application provides a fertile ground for forming very different workflows and lets us demonstrate how diverse the potential applications of our system can be.

6.1 Goals

The features of the system that we will show fall into three categories.

1. **Simplicity of specifying dependencies.** We will show that for simple workflows the dependencies and jobs can be easily submitted using the `mqsub.py` utility. For other workflows, writing the DAG description script is a fairly straightforward task.

2. **Good use of opportunities for concurrent execution.** We introduce the notion of *minimal schedule* and use that notion to argue that TrellisDAG makes good use of the opportunities for concurrency that are inherent in the given workflow.

3. **Small overhead of using the system and placeholder scheduling.** We will quantify the overhead that the user incurs by using TrellisDAG. This overhead includes the overhead of running the placeholders.

The current version of TrellisDAG is not concerned with data movement. Therefore, we factored the data movement out in all of our experiments. This is achieved by predistributing the built checkers endgame databases to all the computation nodes. This does not mean, however, that data transfer cannot be done with the current version. It only means that the user has to take on the responsibility for the data movement and we show how to do it in Section 6.8.2.

6.2 The experiments

We chose to perform three different experiments, from simple to complicated, to show that the system covers the needs of many prospective users and applications:

1. We believe that many computations in practice are simple pipelines and the small-size experiment shows how TrellisDAG can be used to perform a simple pipeline-like computation. We will see that the advanced features of the system (such as the DAG description script) do not have to be used when simpler mechanisms are sufficient. We will also use the small-size experiment to introduce the two kinds of charts that we use to argue that the system makes good use of concurrent execution: the degree of concurrency chart and the resource utilization chart (see Section 6.6).

2. The medium-size experiment shows that the system can be efficiently used to deal with more complicated workflows, such as DAGs. However, these workflows are still simple enough that they do not require the use of TrellisDAG's grouping capability. We will show that there is a trade-off of concurrency versus convenience of monitoring and administration that has to be taken into account when deciding on whether or not to use supergroups. Although defining supergroups may add extra scheduling constraints and negatively affect the degree of concurrency, the convenience benefits could be of higher priority when the workflow is very complicated.

3. The last and largest experiment shows how the system can be used for the most demanding applications. The system was designed with such applica-

tions in mind and provides a mechanism for specifying a hierarchical system of dependencies with the advantage of simplifying the tasks of monitoring and administering the computation. We show that writing the DAG description script for such a computation is not a complicated task either.

6.3 The minimal schedule

In this section we introduce the concept of *minimal schedule*. We use this concept to tackle the following question: "What is a good utilization of opportunities for concurrent execution?" Minimal schedule is a mapping of jobs to resources that is made during a *model run* – an emulated computation where an infinite number of processors is assumed and, whenever all prerequisites of a job are satisfied, the job is run without any start-up overhead.

The algorithm for emulating a model run is as follows:

1. Set time to 0 (i.e. $t = 0$) and the set of running jobs to be empty;
2. Add all the ready jobs to the set of running jobs. For each added job, store its completion time using the time measured during the sequential run. That is, if a job j_1 took t_1 seconds to be computed in the sequential run, then the estimated completion time of j_1 is $t + t_1$;
3. Find the minimal of all completion times in the set of running jobs and set the current time t to that time;
4. Remove those jobs from the set of running jobs whose completion time is t; these jobs are considered complete;
5. If there are more jobs to be executed, goto step 2;

No system that respects the workflow dependencies can beat the model computation as described above in terms of the makespan. Assume, on the contrary, that some schedule S computed a set of jobs J faster than the minimal schedule M. Let $j \in J$ be the first job that was started by S earlier than by M and let j_1 be the last workflow prerequisite of j completed by M. Since j is the first job with the

named property, j_1 could not have been completed earlier by S than by M. Since, M is minimal, it starts computing j without any start-up overhead once it finished computing j_1 and, therefore, it starts j no later than S, resulting in contradiction.

Therefore, if we show that our system results in a schedule that is close to the minimal schedule, we will have shown that the opportunities for concurrency are used well.

6.4 Experimental setup

All of the experiments were performed using the "Jasper" cluster at the Department of Computing Science at the University of Alberta. The cluster consists of a server machine `brule` (dual AMD Athlon 1.5GHz CPUs, 512MB of memory) that is not used for computations and 20 machines with names `jasper-00`, `jasper-01`, ..., `jasper-19` and similar characteristics as the server. The Sun Grid Engine (SGE) software was installed under the author's user account, with the SGE scheduler running on `brule`. The PostgreSQL database engine was running on `jasper-15`, which was also used as a command-line server and not used for computations. To simplify the placeholder and the model for data movement in Section 6.8.2, we used only one processor per machine. Thus we had $\frac{40-2}{2} = 19$ CPUs that could be used for computations.

6.5 The placeholder

In this section, we provide pseudo code for the placeholder used in the experiments. Figure 6.1 shows the high-level algorithmic description of the placeholder. Not shown in the pseudo-code is the fact that all of the services of the command-line server (e.g. steps 2, 3, and 7) are invoked through a retry mechanism. Steps 4 and 6 are concerned with obtaining and sending data to the central host and are only invoked in the experiments with data movement. We explain these operations in Section 6.8.2.

For different reasons, software or hardware, a placeholder may not execute properly (e.g. it can be stopped due to local timing policies). Therefore, it is impor-

1. Run the nanny script in the background.
2. Invoke mqnextjob and get the job ID of the job to run:
   (a) If the returned ID is 0, then resubmit a placeholder and exit.
   (b) If the returned ID is -1, then exit.
3. Invoke mqgetjobcommand and get the command line of the job.
4. Obtain the data from the central host.
5. Run the command line.
6. Send the newly computed data to the central host.
7. Invoke mqdonejob.

Figure 6.1: The layout of the placeholder.

tant to have a fault tolerance mechanism by which TrellisDAG becomes aware of such an event and can schedule the job again on some other machine (in this version of TrellisDAG, such rescheduling requires user assistance).

The *nanny script* implements a simple fault tolerance mechanism. We give the pseudo code for this script in Figure 6.2. The `total time limit` is the maximal time that we allow for any placeholder to run. Once this time is exceeded, the nanny kills the placeholder and exits, so that `mqsignal` is not invoked any more and the system can know that something went wrong. In this version of TrellisDAG, the `total time limit` is set in advance. However, it is possible to use jobs' attributes to set it to a reasonable value for each individual job.

While the total time limit is not exceeded, do:

1. Invoke the mqsignal service with the current time as parameter.
2. Sleep for a specified delay interval.

Kill the placeholder that invoked this script.

Figure 6.2: The nanny script.

The complete placeholder can be found in Appendix A and the nanny script is listed in Appendix B.

6.6 Small-size experiment: 2, 3, and 4 kings

In many cases, the applications of computational science are concerned with performing a multi-stage computation. In such a computation, there are linear (i.e. pipeline) dependencies between jobs – a job can start once the previous job of the pipeline is completed.

The purpose of the small-size experiment is two-fold:

1. Showing that TrellisDAG can be used to submit and compute workflows with pipeline-like dependencies, thereby addressing a common case for the applications of computational science and

2. Introducing the tools to argue the properties of Trellis DAG in the medium-size and large-size experiments. The latter two experiments contain a larger number of jobs with more complicated dependencies than the small-size experiment; we prefer to use a simpler example to explain the resource utilization and degree of concurrency charts – the tools for our argument.

Consider the task of computing the checkers endgame databases for all positions with 2, 3 or 4 kings and no checkers on the board. The DAG for the corresponding workflow is shown in Figure 6.3(a). This translates into the tasks in Figure 6.3(b). Those in turn correspond to the commands in Figure 6.3(c). To demonstrate our point, we ignore the fact that the commands of the lowest rectangle (i.e. 4 kings) in Figure 6.3(c) can be executed concurrently. This workflow is clearly a pipeline, a very common case in real-world applications.

Since there are only three jobs (corresponding to the stages of the pipeline), we do not want to write a DAG description script. We can directly use the `mqsub` utility to submit the jobs. The pipeline in Figure 6.3(c) can be submitted as shown in Figure 6.4.

Note that we did not have to explicitly specify the dependencies between the stages of the pipeline. Since all jobs of the pipeline are contained in one group

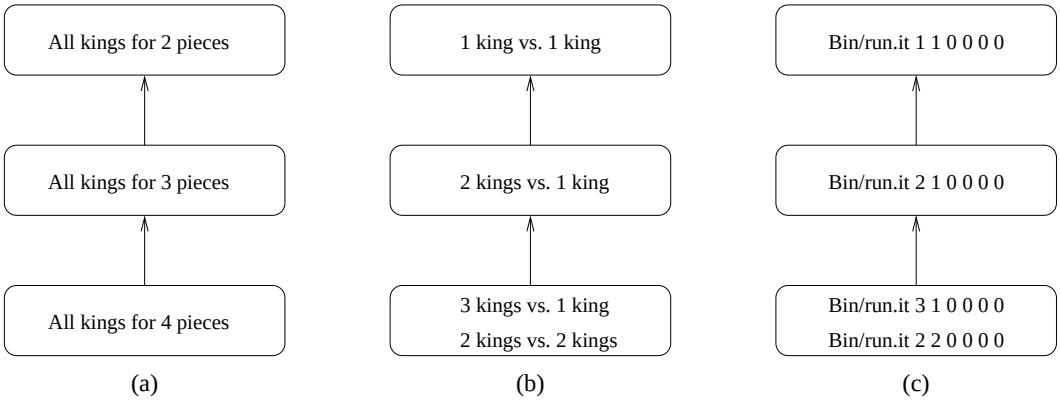

Figure 6.3: The pipeline workflow for computing the endgame databases for up to 4 kings.

```
mqsub -l "pipe0" -deps "" -c "Bin/run.it 1 1 0 0 0 0"
mqsub -l "pipe0" -deps "" -c "Bin/run.it 2 1 0 0 0 0"
mqsub -l "pipe0" -deps "" -c "Bin/run.it 3 1 0 0 0 0; Bin/run.it 2 2 0 0 0 0"
```

Figure 6.4: The submission script for a simple pipeline using implicit dependencies.

`pipe0`, the implicit pipeline dependencies exist and the jobs will be executed in a sequential manner. An alternative way of submitting the same pipeline would be to define a group for each stage of the pipeline. Then the dependencies must be explicitly specified. The submission script is shown in Figure 6.5.

6.6.1 The experiment without placement affinity

We would like to use a variation of the example in Figure 6.4 to demonstrate two kinds of charts that we will use in order to argue some properties of TrellisDAG. The experiment consists of running several copies of the pipeline in Figure 6.3(c) and start these copies with a delay interval from each other. These delays are put in to better explain the charts that we will introduce shortly.

Let us have 3 pairs of copies with 3 seconds delay between the starts of the pairs. We show the experiment schematically in Figure 6.6. The idea is that the pipelines start with a delay, resulting in observing different degrees of concurrency throughout the experiment. The script for running this multi-pipeline experiment is shown in Figure 6.7 (the `$CHECKERS` environmental variable refers to the directory where the placeholder is found).

```
mqsub -l "pipe0" -deps "" -c "Bin/run.it 1 1 0 0 0 0"
mqsub -l "pipe1" -deps "pipe0" -c "Bin/run.it 2 1 0 0 0 0"
mqsub -l "pipe2" -deps "pipe1" -c "Bin/run.it 3 1 0 0 0 0; Bin/run.it 2 2 0 0 0 0"
```

Figure 6.5: The submission script for a simple pipeline using explicit dependencies.

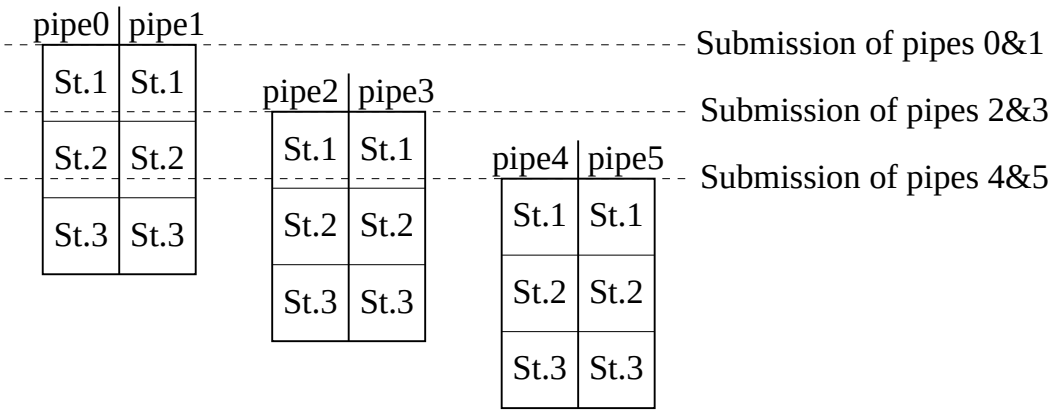

Figure 6.6: The small-size experiment at a glance.

Since there are no dependencies between the six groups (`pipe0`, `pipe1`, `pipe2`, `pipe3`, `pipe4` and `pipe5`), the first job of each group can be started as soon as the corresponding placeholder is submitted (lines 14-15 and 27-28). Note that we only use 4 placeholders and so the maximal possible degree of concurrency is 4.

We run the experiment five times. Table 6.1 shows the makespans obtained in the runs. All of the presented charts correspond to the median run #4. The largest deviation from the makespan from the median run is $\frac{42.82-40.29}{40.29} \approx 0.0628 = 6.28\%$. This deviation is due to several factors:

1. Starting up placeholders by SGE takes different amounts of time (SGE has its own scheduler),

2. Due to dynamic variations of the load of the command-line server, the services of the command line server may take more or less time, and

3. The load of the network varies, affecting the time taken by establishing `ssh` connections.

```
#!/bin/bash

PAUSE=3

# submit pipe0
mqsub -l "pipe0" -deps "" -c "Bin/run.it 1 1 0 0 0 0"
mqsub -l "pipe0" -deps "" -c "Bin/run.it 2 1 0 0 0 0"
mqsub -l "pipe0" -deps "" -c "Bin/run.it 3 1 0 0 0 0; Bin/run.it 2 2 0 0 0 0"
# submit pipe1
mqsub -l "pipe1" -deps "" -c "Bin/run.it 1 1 0 0 0 0"
mqsub -l "pipe1" -deps "" -c "Bin/run.it 2 1 0 0 0 0"
mqsub -l "pipe1" -deps "" -c "Bin/run.it 3 1 0 0 0 0; Bin/run.it 2 2 0 0 0 0"
# submit 2 placeholders
qsub $CHECKERS/template.sh
qsub $CHECKERS/template.sh
sleep $PAUSE #delay

# submit pipe2
mqsub -l "pipe2" -deps "" -c "Bin/run.it 1 1 0 0 0 0"
mqsub -l "pipe2" -deps "" -c "Bin/run.it 2 1 0 0 0 0"
mqsub -l "pipe2" -deps "" -c "Bin/run.it 3 1 0 0 0 0; Bin/run.it 2 2 0 0 0 0"
# submit pipe3
mqsub -l "pipe3" -deps "" -c "Bin/run.it 1 1 0 0 0 0"
mqsub -l "pipe3" -deps "" -c "Bin/run.it 2 1 0 0 0 0"
mqsub -l "pipe3" -deps "" -c "Bin/run.it 3 1 0 0 0 0; Bin/run.it 2 2 0 0 0 0"
# submit 2 placeholders
qsub $CHECKERS/template.sh
qsub $CHECKERS/template.sh
sleep $PAUSE #delay

# submit pipe4
mqsub -l "pipe4" -deps "" -c "Bin/run.it 1 1 0 0 0 0"
mqsub -l "pipe4" -deps "" -c "Bin/run.it 2 1 0 0 0 0"
mqsub -l "pipe4" -deps "" -c "Bin/run.it 3 1 0 0 0 0; Bin/run.it 2 2 0 0 0 0"
# submit pipe5
mqsub -l "pipe5" -deps "" -c "Bin/run.it 1 1 0 0 0 0"
mqsub -l "pipe5" -deps "" -c "Bin/run.it 2 1 0 0 0 0"
mqsub -l "pipe5" -deps "" -c "Bin/run.it 3 1 0 0 0 0; Bin/run.it 2 2 0 0 0 0"
```

Figure 6.7: The submission script for the small-size experiment.

Experiment #	1	2	3	**4**	5
Makespan, sec.	42.82	38.26	42.39	**40.29**	40.15

Table 6.1: The makespans for the small-size experiment. The median run is shown in bold.

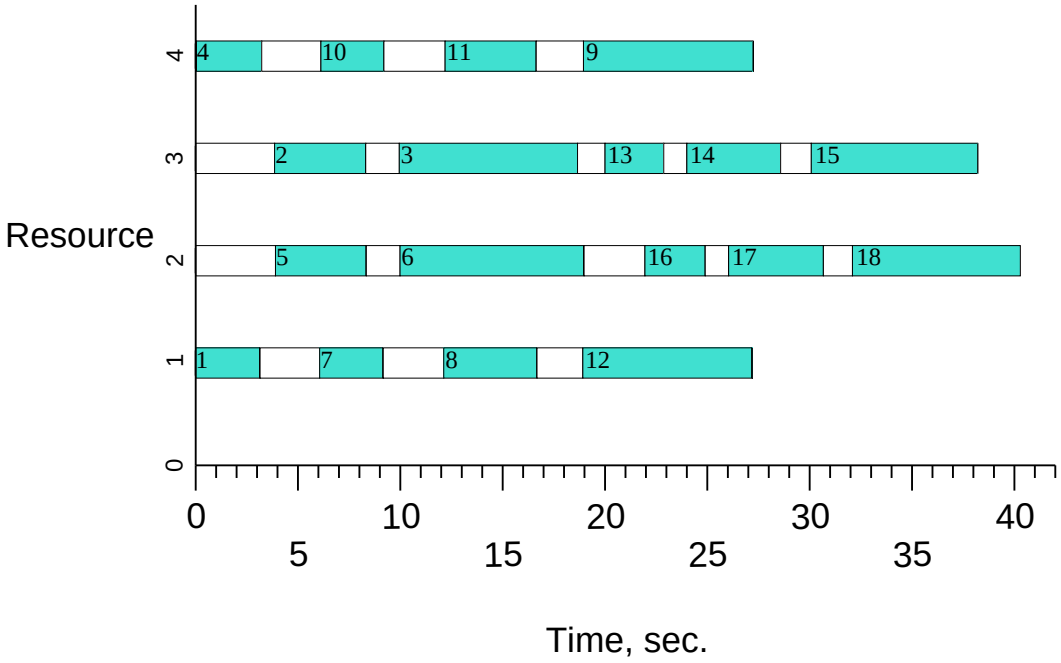

Figure 6.8: Resource utilization chart for the small-size experiment. See Table 6.2 for the definition of jobs.

Resource utilization chart

The results of the experiment are summarized in Figures 6.8 and 6.9. Figure 6.8 is a *resource utilization chart* and shows individual jobs. A job is represented by a green (dark in the b/w copies) bar. It also shows what resources were used to compute particular jobs. For example, Figure 6.8 shows that resource #3 got to execute 5 jobs, but was idle for about 4 seconds at the beginning of the experiment (the reason being that the only jobs available for execution at the beginning of the experiment were taken by the resources #1 and #4), at which point it started executing the job with ID 2.

The corresponding job's ID is provided together with each bar to improve the readability of the resource utilization chart; furthermore, we provide a table by which the correspondence of ID's to jobs is established. For our example, this is Table 6.2.

Table 6.2 also shows how much time was used by the placeholders and how much of that time was due to the overhead (both SGE and TrellisDAG). Note that

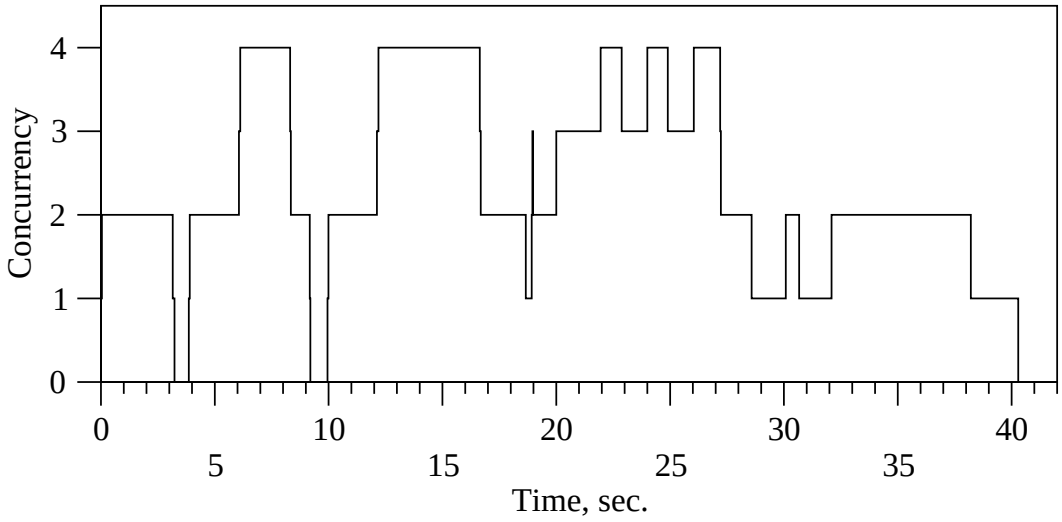

Figure 6.9: Degree of concurrency chart for the small-size experiment.

the overhead time is quite large compared to the computation time. In other words, the granularity of our experiment is low. However, the computation-to-overhead ratio was sufficient for our experiment to achieve a speedup. The time for a sequential run can be obtained by summing up all of the computation times. Hence, it would take 64.16 seconds to execute this experiment on a single workstation of our cluster. Our experiment, on the other hand, took 40.29 seconds (see Table 6.1) and would certainly finish earlier if there were no delays between the starting times of the pipelines.

Degree of concurrency chart

Figure 6.9 is a *degree of concurrency chart* and shows how the degree of concurrency changed over time during the experiment. Note that the same information can be derived from the resource utilization chart, but the degree of concurrency chart makes the trends easier to see. As an example of how to interpret a degree of concurrency chart, Figure 6.9 shows that the concurrency level of 4 was first achieved after about 6 seconds from the start of the experiment.

Observations

Note that when the first placeholder requested a job when the experiment just started, there was a choice between 2 jobs that were ready for execution: the jobs

Job#	Job name	Computation time, sec	Other time, (overhead), sec	Total time, sec
1	2 kings of `pipe0`	1.35	2.67	4.02
2	3 kings of `pipe0`	2.82	2.50	5.32
3	4 kings of `pipe0`	6.54	3.09	9.63
4	2 kings of `pipe1`	1.36	2.67	4.03
5	3 kings of `pipe1`	2.80	2.52	5.32
6	4 kings of `pipe1`	6.53	3.36	9.89
7	2 kings of `pipe2`	1.35	2.61	3.96
8	3 kings of `pipe2`	2.83	2.55	5.38
9	4 kings of `pipe2`	6.51	2.88	9.39
10	2 kings of `pipe3`	1.36	2.56	3.92
11	3 kings of `pipe3`	2.82	2.45	5.27
12	4 kings of `pipe3`	6.51	2.98	9.49
13	2 kings of `pipe4`	1.34	2.40	3.74
14	3 kings of `pipe4`	2.84	2.54	5.38
15	4 kings of `pipe4`	6.54	2.40	8.94
16	2 kings of `pipe5`	1.34	2.40	3.74
17	3 kings of `pipe5`	2.81	2.63	5.44
18	4 kings of `pipe5`	6.51	2.45	8.96
$\sum$	Total sum	64.16	47.66	111.82

Table 6.2: Time of computation and overhead for the small-size experiment. Job# corresponds to annotations in Figure 6.8.

with IDs 1 and 4. Our scheduler simply chooses the job with minimal job ID. However, any other scheduling policy could be implemented (see Section 7.1).

We now address two questions about the experiment:

1. Why would one want to separate a pipeline into individual jobs? and
2. After the separation is done, why not run all of the jobs of the pipeline on the same machine? After all, the data produced by a job of the pipeline will likely be needed by the next job and we can avoid data migration by running the jobs on the same host.

We identify three reasons:

1. Although it was not the case in our experiment, there may be dependencies between pipelines. Consider, for example, a multi-pipeline workflow in which the first stage of a pipeline can only be executed after the second stage of another pipeline is completed. In that case, we need to split the latter pipeline into at least two parts (provided, of course, that this pipeline has more than two stages),
2. The intermediate results of the pipeline can be of value. The points of a workflow at which intermediate results are of value are called *end-points* (sometimes end-points are also exit nodes, but not always as in this example with pipelines). For example, the user may want to look at the results of the computation for 2 kings as soon as this result is available. Although none of the currently implemented utilities of our system can give a per-job information on the status of execution, the mechanisms are in place and such a utility can be added in the next version of TrellisDAG, and
3. There may be a scheduling benefit in running the jobs of the pipeline on different machines and we show this experimentally in Section 6.6.2.

6.6.2 The experiment with placement affinity

In the previous section, we gave three reasons for separating a pipeline into several independent jobs that can run on different machines. In particular, we claimed that

Experiment #	**1**	2	3	4	5
Makespan, sec.	**42.59**	42.90	43.22	42.49	42.51

Table 6.3: The makespans for the small-size experiment with placement affinity. The median run is shown in bold.

there may be a scheduling benefit in running the jobs of a pipeline on different machines. In this section, we provide justification for this claim.

Let us force the jobs of each of our pipelines to run on the same host. We use the `affinity` attribute to do that. The new submission script is shown in Figure 6.10.

We ran the experiment five times. Table 6.3 shows the makespans obtained in the runs. All of the presented charts correspond to the median run #1. The largest deviation from the makespan of the median run is $\frac{43.22-42.59}{42.59} \approx 0.0148 = 1.48\%$.

The resource utilization chart and the degree of concurrency chart for this experiment are shown in Figures 6.12 and 6.13, respectively.

Let us concentrate on the resource utilization chart in Figure 6.12. There are two resources (#1 and #3) that each executed 6 jobs, while the highest number of jobs per resource in Figure 6.8 is 5. This is not a coincidence. In the present experiment, there are 6 pipelines and 4 resources; hence, by the pigeon-hole principle, at least one of the resources has to execute 2 pipelines because the pipelines are not split. As a result, the computation took longer than the computation where stages of the pipeline could be executed on different computational nodes.

The situation is illustrated in Figure 6.11. There are 6 pipelines and 4 resources. With placement affinity as in our example, all stages of a pipeline must be scheduled to one resource and, in effect, a pipeline is scheduled as one unit. Assume some mapping of pipelines with numbers 0, 1, 2 and 3 to resources. Then, there is no way of mapping the remaining pipelines without having one resource compute all jobs of more than one pipeline.

```
 1  #!/bin/bash
 2
 3  PAUSE=3
 4
 5  mqsub -l "pipe0" -deps "" -c "Bin/run.it 1 1 0 0 0 0"
 6  mqsub -l "pipe0" -deps "" -c "Bin/run.it 2 1 0 0 0 0" \
 7        -attr "affinity=yes"
 8  mqsub -l "pipe0" -deps "" -c "Bin/run.it 3 1 0 0 0 0; Bin/run.it 2 2 0 0 0 0" \
 9        -attr "affinity=yes"
10  mqsub -l "pipe1" -deps "" -c "Bin/run.it 1 1 0 0 0 0"
11  mqsub -l "pipe1" -deps "" -c "Bin/run.it 2 1 0 0 0 0" \
12        -attr "affinity=yes"
13  mqsub -l "pipe1" -deps "" -c "Bin/run.it 3 1 0 0 0 0; Bin/run.it 2 2 0 0 0 0" \
14        -attr "affinity=yes"
15  qsub $CHECKERS/template.sh
16  qsub $CHECKERS/template.sh
17  sleep $PAUSE
18
19  mqsub -l "pipe2" -deps "" -c "Bin/run.it 1 1 0 0 0 0"
20  mqsub -l "pipe2" -deps "" -c "Bin/run.it 2 1 0 0 0 0" \
21        -attr "affinity=yes"
22  mqsub -l "pipe2" -deps "" -c "Bin/run.it 3 1 0 0 0 0; Bin/run.it 2 2 0 0 0 0" \
23        -attr "affinity=yes"
24  mqsub -l "pipe3" -deps "" -c "Bin/run.it 1 1 0 0 0 0"
25  mqsub -l "pipe3" -deps "" -c "Bin/run.it 2 1 0 0 0 0" \
26        -attr "affinity=yes"
27  mqsub -l "pipe3" -deps "" -c "Bin/run.it 3 1 0 0 0 0; Bin/run.it 2 2 0 0 0 0" \
28        -attr "affinity=yes"
29  qsub $CHECKERS/template.sh
30  qsub $CHECKERS/template.sh
31  sleep $PAUSE
32
33  mqsub -l "pipe4" -deps "" -c "Bin/run.it 1 1 0 0 0 0"
34  mqsub -l "pipe4" -deps "" -c "Bin/run.it 2 1 0 0 0 0" \
35        -attr "affinity=yes"
36  mqsub -l "pipe4" -deps "" -c "Bin/run.it 3 1 0 0 0 0; Bin/run.it 2 2 0 0 0 0" \
37        -attr "affinity=yes"
38  mqsub -l "pipe5" -deps "" -c "Bin/run.it 1 1 0 0 0 0"
39  mqsub -l "pipe5" -deps "" -c "Bin/run.it 2 1 0 0 0 0" \
40        -attr "affinity=yes"
41  mqsub -l "pipe5" -deps "" -c "Bin/run.it 3 1 0 0 0 0; Bin/run.it 2 2 0 0 0 0" \
42        -attr "affinity=yes"
```

Figure 6.10: The submission script for the small-size experiment with placement affinity.

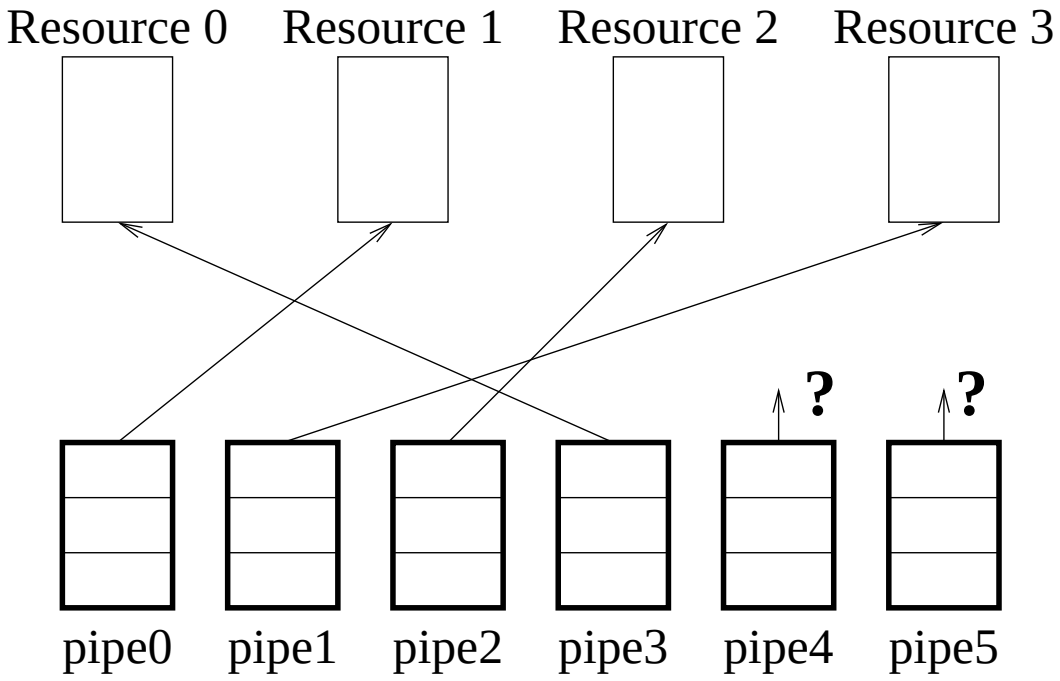

Figure 6.11: Placement affinity and pigeon-hole principle.

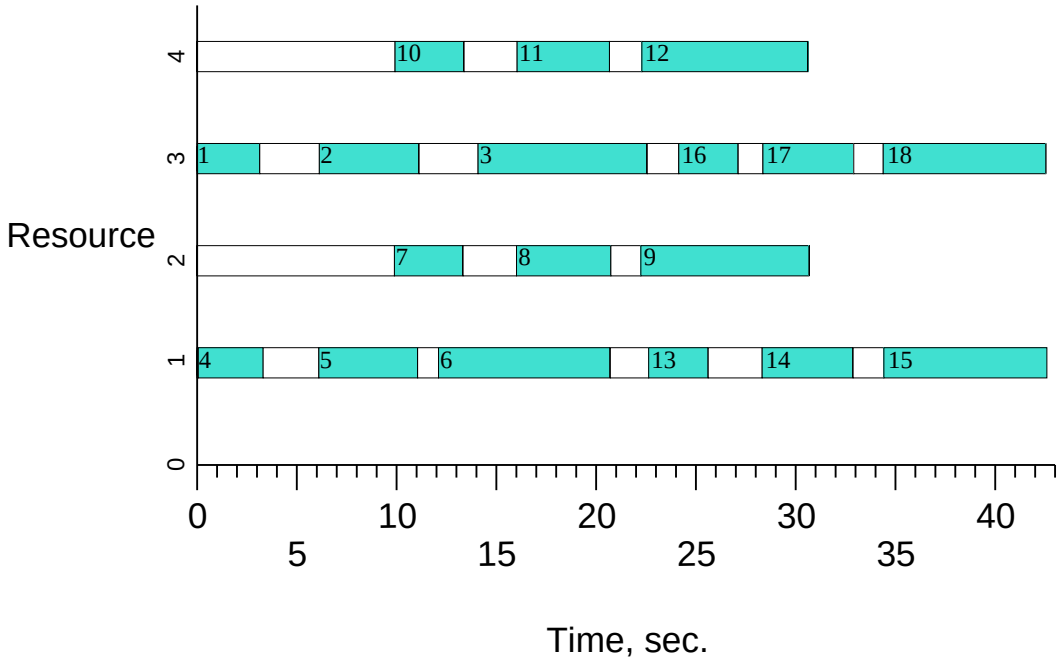

Figure 6.12: Resource utilization chart for the small-size experiment with placement affinity. See Table 6.2 for the definition of jobs.

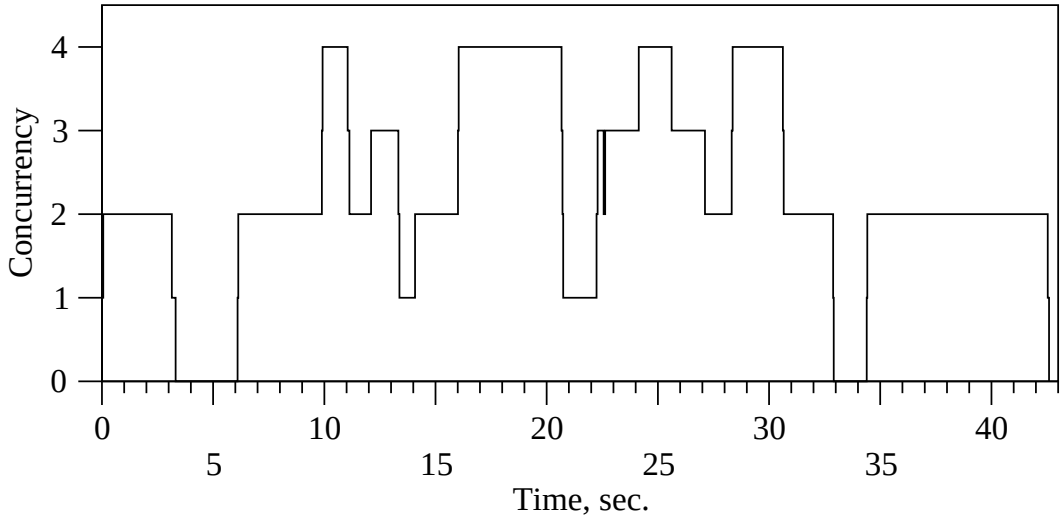

Figure 6.13: Degree of concurrency chart for the small-size experiment with placement affinity.

6.6.3 Summary

Although TrellisDAG is specifically designed for applications that exhibit complicated workflows with a multitude of jobs and dependencies, it is well suited for computing simple pipeline workflows. We showed that there are reasons (among which are scheduling issues) that can justify splitting a large job into several jobs with pipeline dependencies and even allowing those jobs to execute on different machines. By the way of example, we introduced two charts – the resource utilization chart and the degree of concurrency chart – that we will use as tools for arguing the properties of TrellisDAG in the following sections.

6.7 Medium-size experiment: all-kings databases up to 7 pieces

In this section, we show how the system can be used to execute a workflow that is easily represented as a plain DAG. The example that we use is all-kings checkers databases for positions with up to and including 7 pieces on the board. Consistent with the explanation in Chapter 3, the DAG for our computation is shown in Figure 6.14. Note that the marked part of this DAG was used as a single pipeline for the small-size experiment.

There are 12 groups (of 2 jobs each) and 15 dependencies in the DAG in Figure 6.14. Note that it can be viewed as a pipeline of 6 stages (see Figure 6.15(a)), where all groups in each stage can be computed in parallel. Therefore, it is natural to define supergroups that represent the individual stages of the pipeline. Then, the workflow looks as shown in Figure 6.15(a). However, this workflow contains many implicit dependencies and is, in fact, equivalent to the workflow in Figure 6.15(b). The reader can see that this DAG has more dependencies compared to the DAG shown in Figure 6.14. For example, the slice with 6 black kings and 1 white king can be computed once the slice with 5 black kings and 1 white king is computed; however, with the supergroups defined, the aforementioned slice cannot be started until all of the slices with 6 kings on the board are computed.

We will show both how to run the computation with the workflow shown in

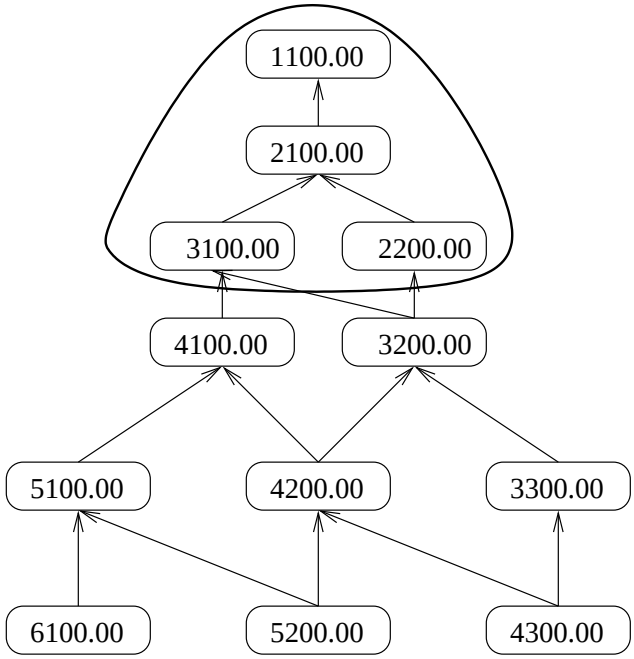

Figure 6.14: The DAG for the all-kings computation.

Experiment #	1	2	**3**	4	5	Minimal schedule
Makespan, sec.	4005.88	4035.56	**4020.09**	4004.03	4024.54	3988.85

Table 6.4: The makespans for the medium-size experiment. The median run is shown in bold.

Figure 6.14 and how to join the stages of the pipeline into supergroups and run the computation with the workflow shown in Figure 6.15. Then we will conclude as to when the grouping capability should be used.

6.7.1 The experiment without supergroups

The DAG description script for this computation is is shown in Figure 6.16.

We run the experiment five times. Table 6.4 shows the makespans obtained in the runs. All of the presented charts correspond to the median run #3. The largest deviation from the makespan of the median run is $\frac{4020.09-4004.03}{4020.09} \approx 0.0040 = 0.4\%$.

The flow of computation is summarized with the resource utilization chart in Figure 6.17 and the degree of concurrency chart in Figure 6.20. Because of the

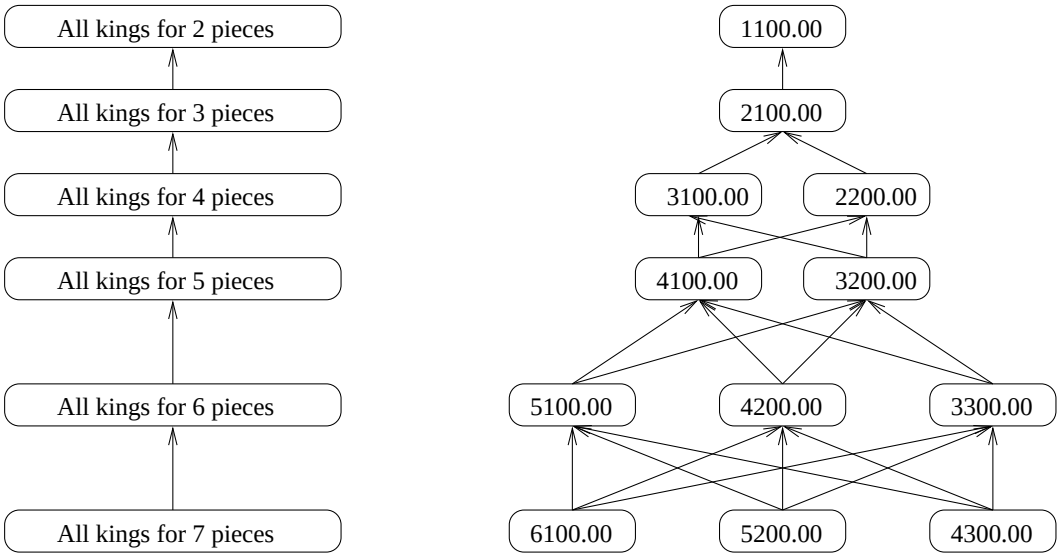

Figure 6.15: Using the grouping capability for the all-kings computation.

low granularity of jobs at the beginning of the experiment, we include Figures 6.18, 6.19, 6.21 and 6.22 that scale the portions of Figures 6.17 and 6.20 corresponding to the beginning of the experiment. We noticed that the degree of concurrency in this computation never exceeded 5 jobs. Therefore, in order to keep the graph simple, we enabled only 5 SGE queues (each SGE queue corresponding to one machine) and used 5 placeholders. As we see, some of the jobs are very short and some of the jobs at the end of the computation were of considerable length. Of course, this is a property of the application domain and not the schedule. The relative overhead of using our system is different for such jobs. Table 6.5 summarizes the times used for computation and the measured overhead.

Figure 6.20 shows that most of the time the degree of concurrency was 1. We have to note that this results from the inherent limitations of the application rather than our system. Specifically, the executions of the two longest jobs with IDs 35 and 36 cannot be overlapped, because job #36 is a verification stage of the computation made by job #35 (see Table 6.5).

Job#	Job name	Computation/ verification time, sec.	Other time (overhead), sec.	Total time, sec.
2	1100.00 computation	1.35	2.81	4.16
3	1100.00 verification	2.71	2.35	5.06
5	2100.00 computation	2.85	2.78	5.63
6	2100.00 verification	3.00	2.32	5.32
8	3100.00 computation	3.82	2.67	6.49
9	3100.00 verification	2.92	2.91	5.83
11	2200.00 computation	2.76	2.64	5.40
12	2200.00 verification	4.10	2.31	6.41
14	4100.00 computation	10.36	2.92	13.28
15	4100.00 verification	4.57	2.43	7.00
17	3200.00 computation	45.19	2.86	48.05
18	3200.00 verification	15.56	2.32	17.88
20	5100.00 computation	44.12	2.59	46.71
21	5100.00 verification	13.14	2.33	15.47
23	4200.00 computation	124.05	2.63	126.68
24	4200.00 verification	86.47	2.29	88.76
26	3300.00 computation	154.77	2.44	157.21
27	3300.00 verification	103.11	2.31	105.42
29	6100.00 computation	188.72	2.37	191.09
30	6100.00 verification	47.80	2.26	50.06
32	5200.00 computation	688.47	2.31	690.78
33	5200.00 verification	328.72	2.27	330.99
35	4300.00 computation	2952.50	2.36	2954.86
36	4300.00 verification	828.36	2.34	830.76
$\sum$	Total sum	5659.42	59.82	5719.24

Table 6.5: Time of computation and overhead for the medium-size experiment.

```
 1  #!/usr/bin/python
 2
 3  import sys
 4  import math
 5  import string
 6
 7  nPieces = int(sys.argv[1])
 8
 9  def isDependentRank(rank1, rank2):
10      if rank1[0] >= rank2[0] and rank1[1] >= rank2[1]:
11          return 1
12      return 0
13
14  #THE INTERFACE PART
15  Groups = [['Rank', 6]]
16
17  def generateGroup(level, parentGroup):
18      result = []
19      for np in range(2, nPieces + 1):
20          for b in range(1, np/2 + 1):
21              result.append([np - b, b, 0, 0, 0, 0])
22      return result
23
24  def getPrologueExecutables(level, group): return []
25  def getPrologueAttributes(level, group): return []
26  def getPrologueParameters(level, group): return []
27  def getEpilogueExecutables(level, group): return []
28  def getEpilogueAttributes(level, group): return []
29  def getEpiloguеParameters(level, group): return []
30
31  def getJobsExecutables(level, group):
32      return ["Bin/run.it", "Bin/ver.it"]
33
34  def getJobsAttributes(level, group):
35      return [["release=" + "yes"], ["release=" + "no"]]
36
37  def getJobsParameters(level, group):
38      return [string.join(map(str, group), " "),
39              string.join(map(str, group), " ")]
40
41  def isDependent(level, group1, group2, parentGroup):
42      return isDependentRank(group1, group2)
```

Figure 6.16: The DAG description script for the all-kings computation.

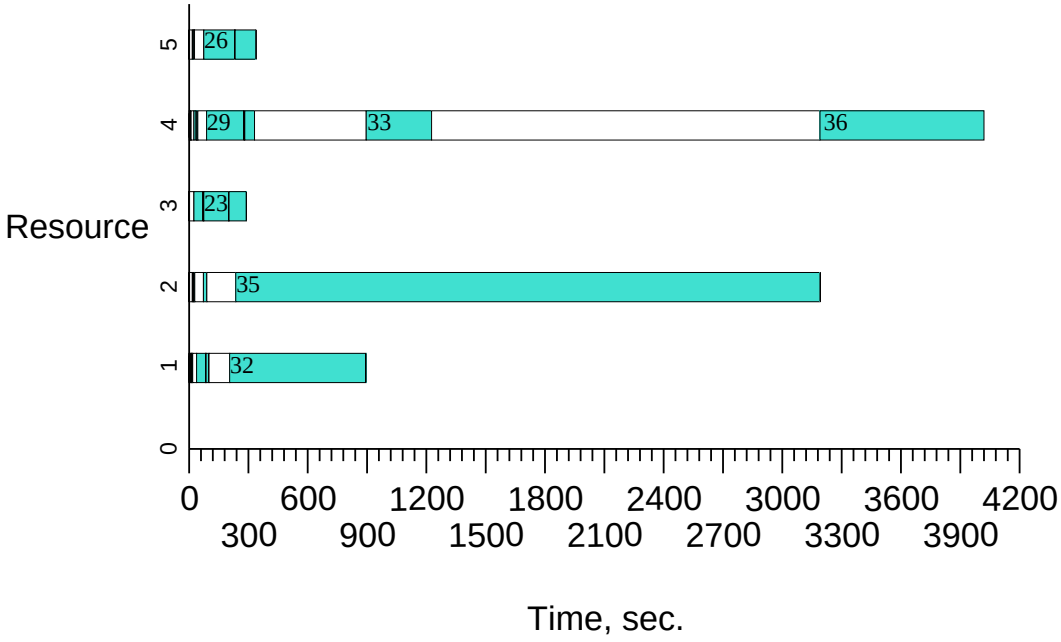

Figure 6.17: Resource utilization chart for the medium-size experiment. See Table 6.5 for the definition of jobs.

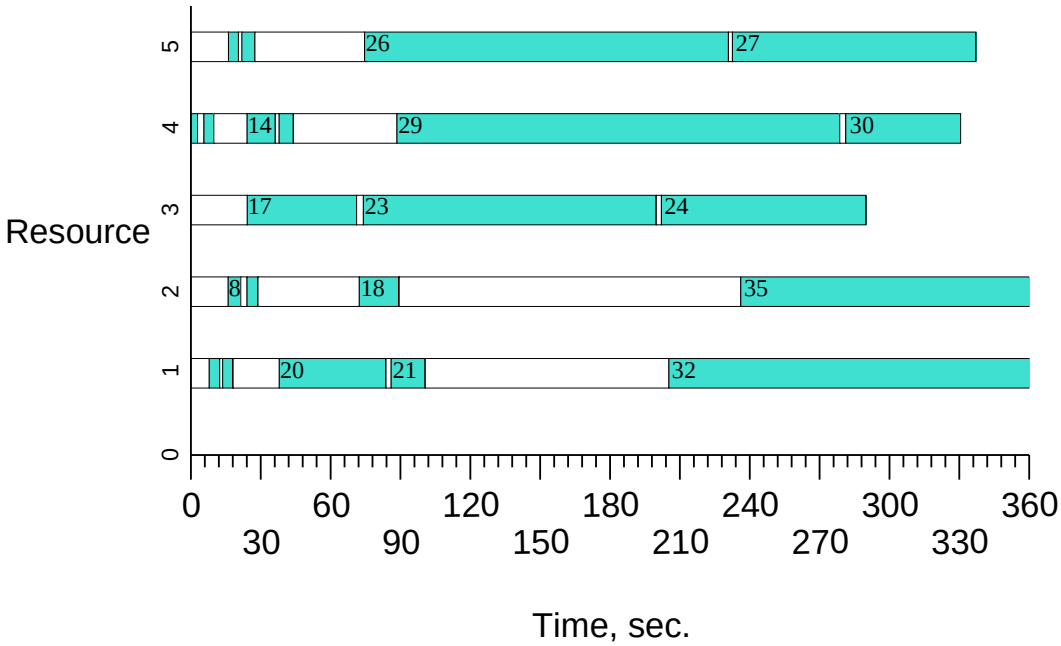

Figure 6.18: Resource utilization chart for the medium-size experiment. This is a scaled version of the beginning part of the chart in Figure 6.17. See Table 6.5 for the definition of jobs.

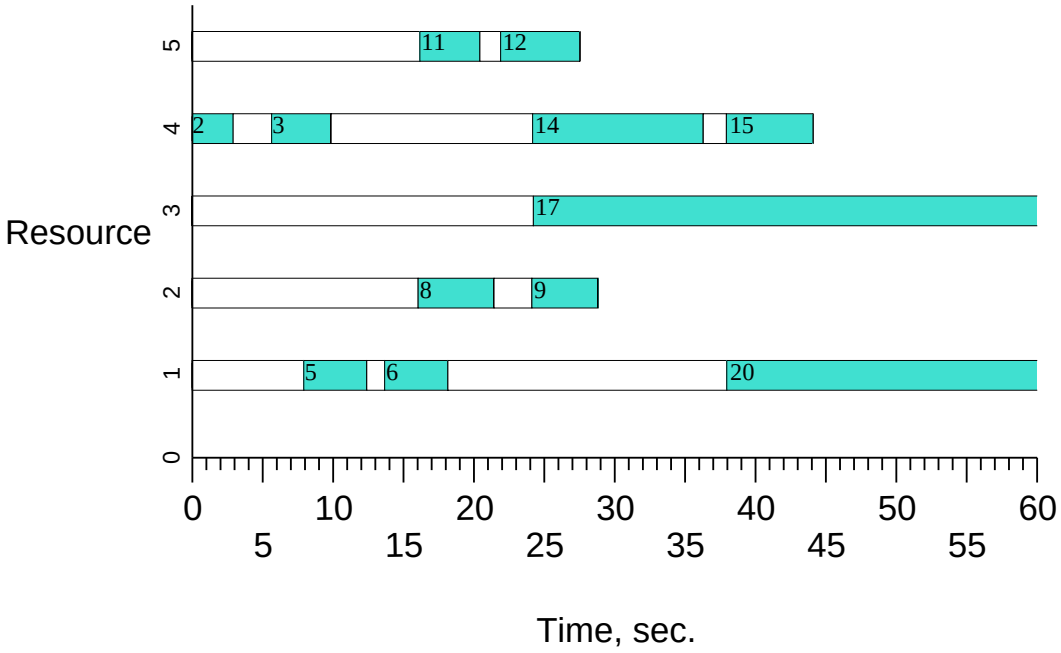

Figure 6.19: Resource utilization chart for the medium-size experiment. This is a scaled version of the beginning part of the chart in Figure 6.18. See Table 6.5 for the definition of jobs.

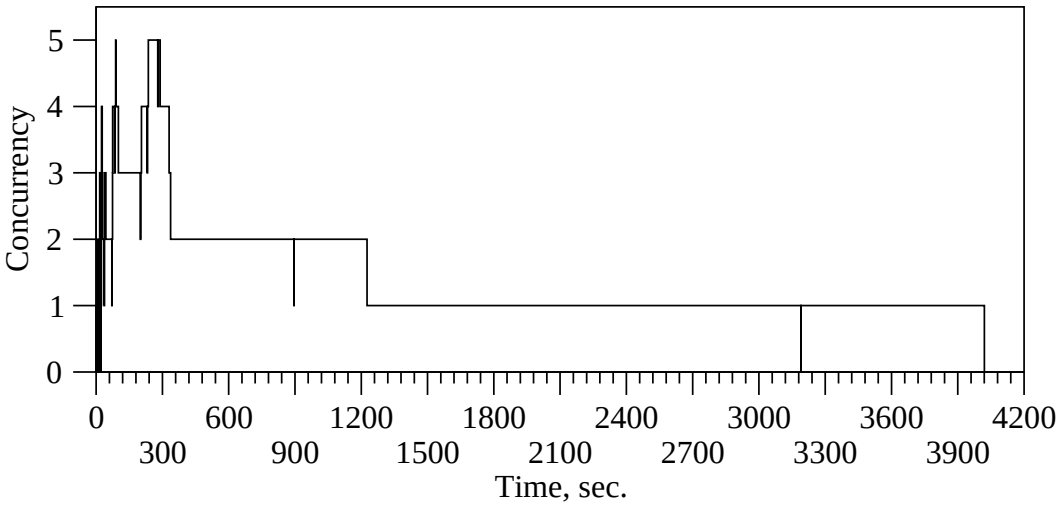

Figure 6.20: Degree of concurrency chart for the medium-size experiment.

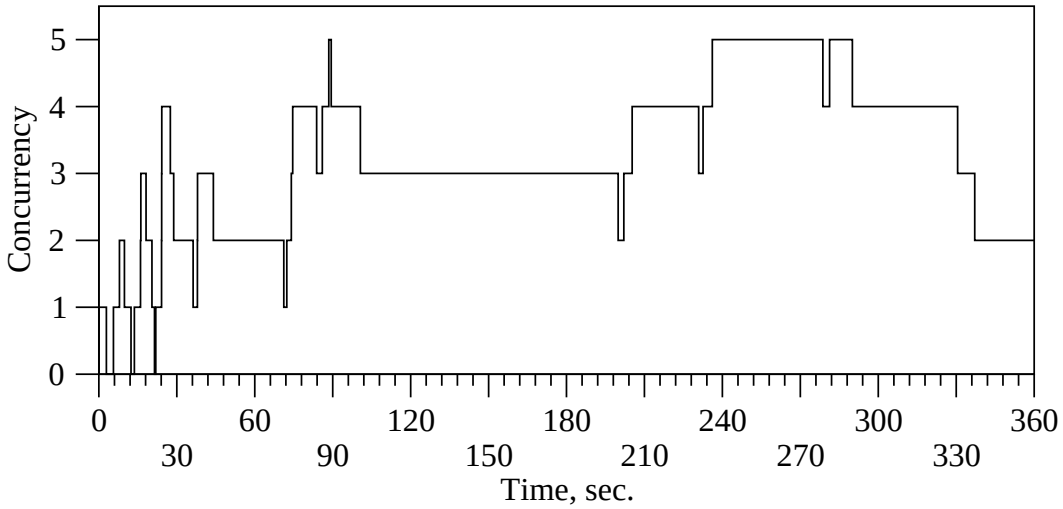

Figure 6.21: Degree of concurrency chart for the medium-size experiment. This is a scaled version of the beginning part of the chart in Figure 6.20.

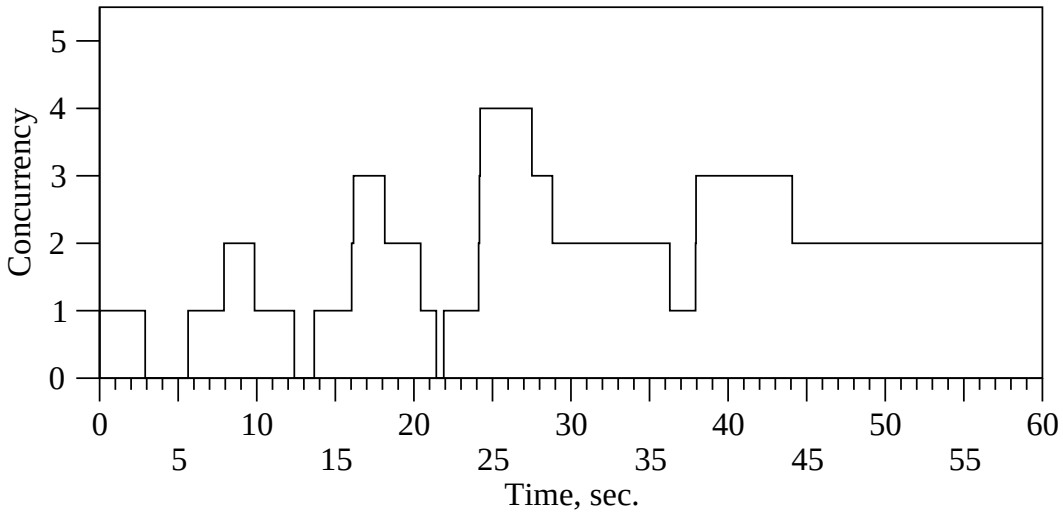

Figure 6.22: Degree of concurrency chart for the medium-size experiment. This is a scaled version of the beginning part of the chart in Figure 6.21.

6.7.2 The use of opportunities for concurrency

We will now compare the computation in Section 6.7.1 to the one that results from the minimal schedule. The resource utilization chart for the model run is shown in Figure 6.23. The lines over the bars show when the corresponding jobs started and stopped in during the real run. Figures 6.24 and 6.25 show the scaled versions of the chart.

We first concentrate on Figure 6.25. First look at job #2. We see that it took more time in the real run than in the minimal schedule. The reason is that the running time in the real time is measured from the time when the job is inserted into the `Running` table to the time when it is deleted from that table. In contrast, the minimal schedule only takes into account the time taken to run the executable itself.

The minimal schedule shows that the jobs #3 and #5 can be started simultaneously without violating any dependencies. However, in the real run, job #5 started well after job #3 started. The main reason of such delay is the scheduling delay of SGE. As can be seen in the placeholder (see Appendix A), we save the time when a placeholder starts and when a placeholder finishes. We found that the placeholder that executed job #5 started after a significant delay from the moment when the placeholder that submitted this placeholder finished.

Now, let us look at Figure 6.23. We note that the last job with ID 36 started not much later in the real run compared to the minimal schedule. This seems strange after we saw big differences in Figure 6.25. However, we have to note that the overhead accumulates only on the paths that go from the entry nodes of the DAG to the exit nodes (see Figure 6.14). For job #36, this corresponds to more than one path. However, the overheads of the paths are not added; instead, the maximum of the overheads of the paths should be taken. Likely, this is the overhead that results from a maximal (in terms of the number of nodes) path. For example, the following path is maximal in our case: `1100.00 -> 2100.00 -> 2200.00 -> 3200.00 -> 3300.00 -> 4300.00` with length 7. Thus, only 7 jobs (including the verification stage of 4300.00) constitute most to the delay of the last job. The result is that model computation would take 3988.85 seconds to run, while

the real computation took 4020.09 seconds which is only about 0.78% slower than the model experiment.

The degree of concurrency chart and its scaled versions are presented in Figures 6.26, 6.27 and 6.28. The dashed graph represents the corresponding degrees of concurrency of the real run. The general patterns of the two runs shown in Figure 6.26 are very close. When we look at more detailed Figures 6.27 and 6.28, we see that there are differences, but the minimal schedule does not have a clear advantage over the real run in terms of the achieved degrees of concurrency.

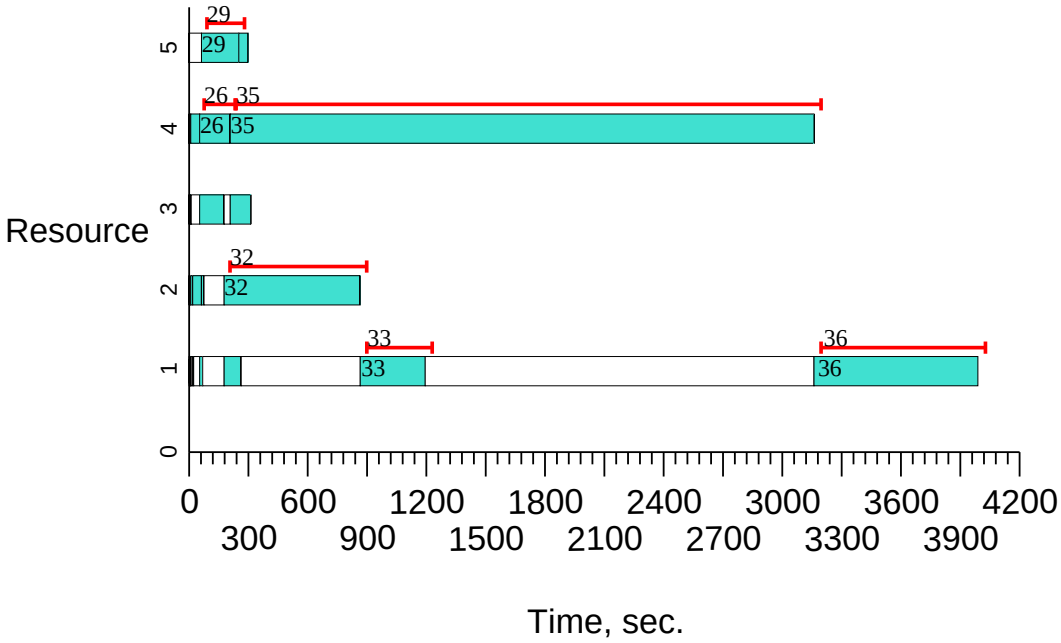

Figure 6.23: Resource utilization chart for the medium-size experiment. Model run. The lines over bars represent the real run (see Figures 6.17, 6.18, and 6.19). See Table 6.5 for the definition of jobs.

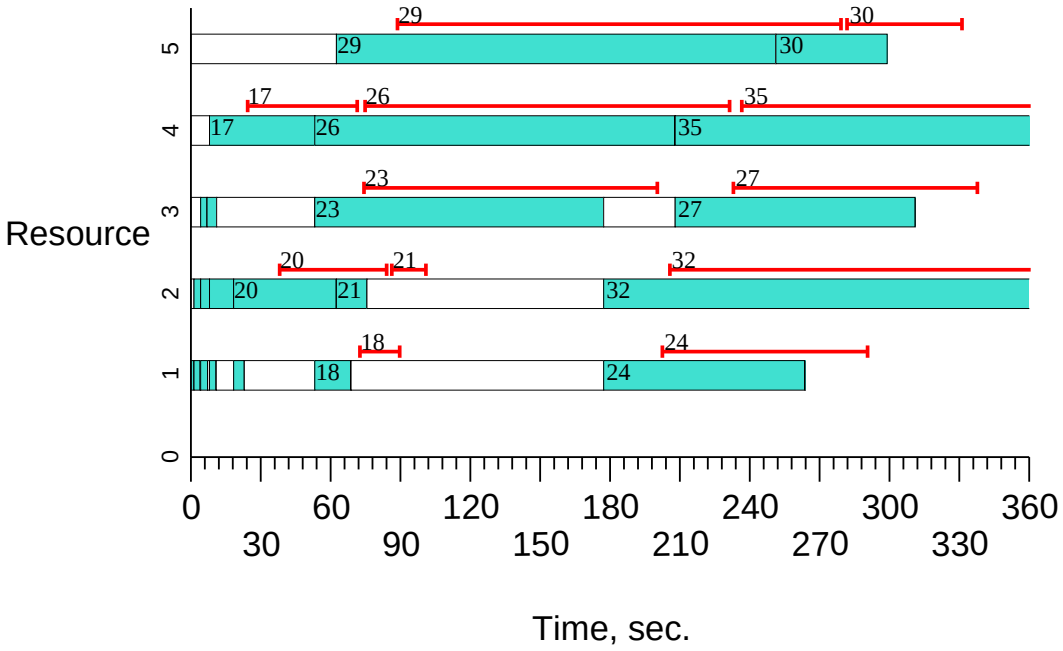

Figure 6.24: Resource utilization chart for the medium-size experiment. Model run. This is a scaled version of the beginning part of the chart in Figure 6.23. The lines over bars represent the real run (see Figures 6.17, 6.18, and 6.19). See Table 6.5 for the definition of jobs.

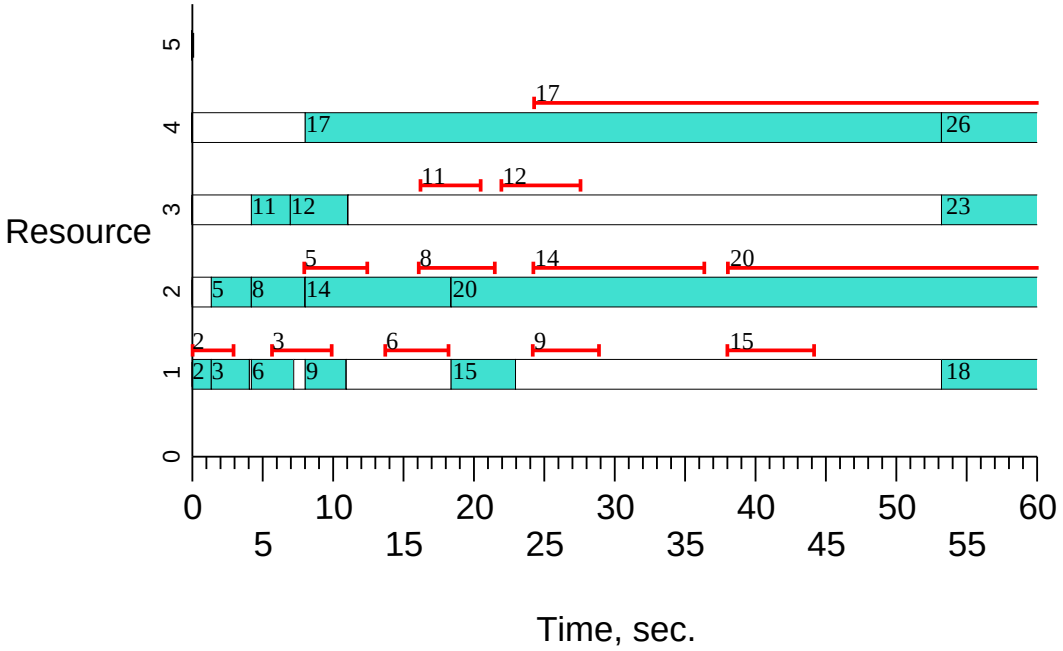

Figure 6.25: Resource utilization chart for the medium-size experiment. Model run. This is a scaled version of the beginning part of the chart in Figure 6.24. The lines over bars represent the real run (see Figures 6.17, 6.18, and 6.19). See Table 6.5 for the definition of jobs.

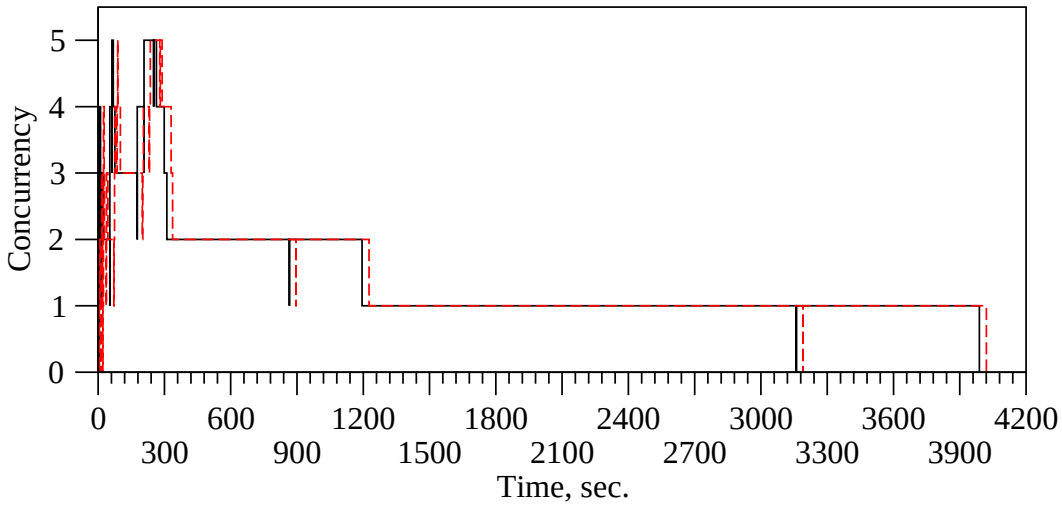

Figure 6.26: Degree of concurrency chart for the medium-size experiment. Model run. The dashed graph represents the corresponding levels of concurrency for the real run (see Figures 6.20, 6.21, and 6.22).

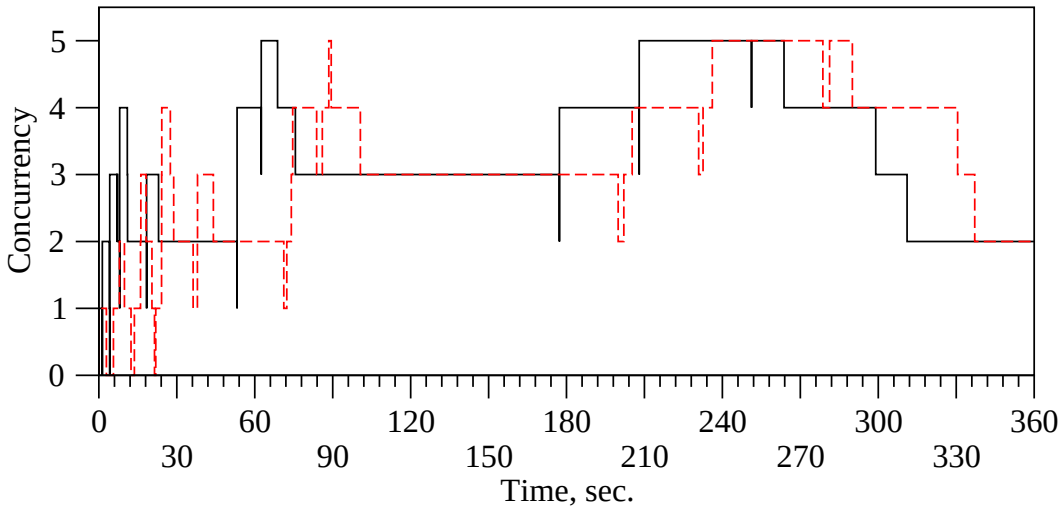

Figure 6.27: Degree of concurrency chart for the medium-size experiment. Model run. This is a scaled version of the beginning part of the chart in Figure 6.26. The dashed graph represents the corresponding levels of concurrency for the real run (see Figures 6.20, 6.21, and 6.22).

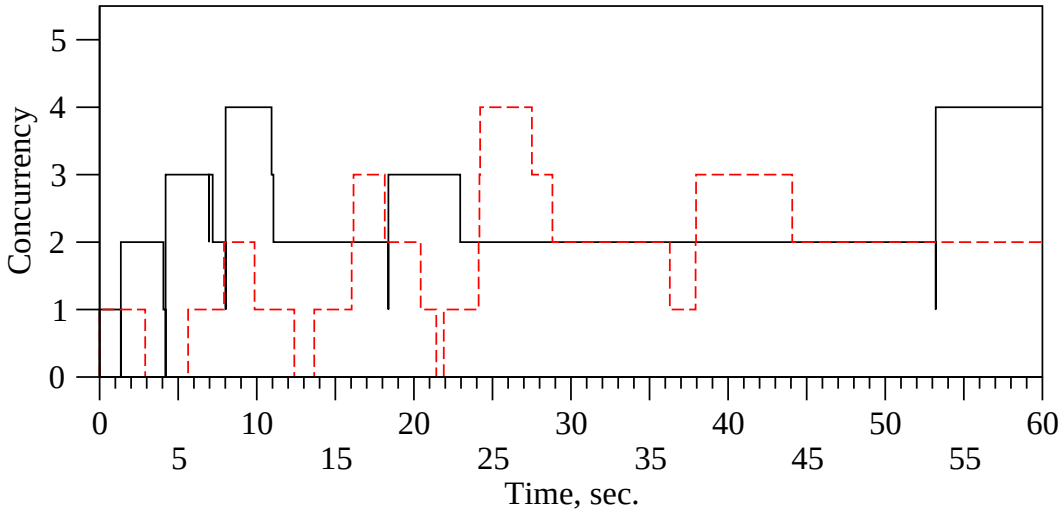

Figure 6.28: Degree of concurrency chart for the medium-size experiment. Model run. This is a scaled version of the beginning part of the chart in Figure 6.27. The dashed graph represents the corresponding levels of concurrency for the real run (see Figures 6.20, 6.21, and 6.22).

Experiment #	1	2	3	**4**	5	Minimal schedule
Makespan, sec.	4034.93	4029.07	4070.02	**4035.57**	4064.66	4002.35

Table 6.6: The makespans for the medium-size experiment with supergroups. The median run is shown in bold.

6.7.3 The experiment with supergroups

The DAG description script for running the all-kings computation with supergroups is shown in Figure 6.29.

We run the experiment five times. Table 6.6 shows the makespans obtained in the runs. All of the presented charts correspond to the median run #4. The largest deviation from the makespan of the median run is $\frac{4070.02-4035.57}{4035.57} \approx 0.0085 = 0.85\%$.

The resource utilization chart for the computation is shown in Figure 6.30. The lines over the bars show when the corresponding jobs started and stopped during the run without supergroups. Figures 6.31 and 6.32 show the scaled versions of the chart.

At the first glance, the schedules for the runs with and without supergroups do not differ much. Suppose, however, that the investigators of the checkers project are interested in the databases for the cases of 4 kings versus 3 kings, 5 kings versus 2 kings, and 6 kings versus 1 king (i.e. three end-points). We compare the times of achieving these end-point results with and without using supergroups. This comparison is summarized in Table 6.7. We see that, although in our example the makespan of the computation is almost the same without supergroups and with supergroups (the difference between the makespans of the median runs being $4035.57 - 4020.09 = 15.48$ seconds, see Tables 6.4 and 6.6), certain end-points can be achieved faster without supergroups.

The degree of concurrency chart and its scaled versions are presented in Figures 6.33, 6.34 and 6.35. The dashed graph represents the corresponding degrees of concurrency of the run without supergroups.

One noticeable difference between the computations without supergroups and

Job# w/o super-groups	Job# with super-groups	Job name	Reached w/o supergroups, sec	Reached with supergroups, sec
29	40	6100.00 computation	278.66	428.14
32	43	5200.00 computation	895.27	920.09
35	46	4300.00 computation	3189.28	3202.85

Table 6.7: Comparison of the times of reaching the end-points of the all-kings computation with and without supergroups.

with supergroups is seen in Figure 6.35. We see that, after 44 seconds of execution, the degree of concurrency stays at 2 without supergroups and at 1 with supergroups. The reason becomes clear when we look at Figure 6.31. Job with ID 29 started earlier in the run without supergroups and contributed to the degree of concurrency of the mentioned time interval. The job could start earlier because it did not depend on the jobs for the 5-piece databases, while when we use supergroups, all of the 5-piece (and, in particular, the job with ID 24) databases have to be computed before any job for the 6-piece databases can start.

We conclude that there is a trade-off when using the TrellisDAG's grouping capability. Once the supergroups are defined, the user is provided with tools to monitor the execution in a convenient way. However, the scope of opportunities for concurrency used by the system will possibly be below optimal. In the all-kings computation, the total execution time is not affected by introducing supergroups and corresponding extra dependencies. However, if we take the standpoint that the faster time of completion of individual jobs increases user's satisfaction, then it is beneficial to not define supergroups. After all, the workflow in this experiment is pretty simple and monitoring will not be too cumbersome even without super-groups.

Job#	Job name	Computation/ verification time, sec.	Other time (overhead), sec.	Total time, sec.
3	1100.00 computation	1.33	2.82	4.15
4	1100.00 verification	2.69	2.36	5.05
8	2100.00 computation	2.85	2.91	5.76
9	2100.00 verification	3.13	2.35	5.48
13	3100.00 computation	3.90	2.70	6.60
14	3100.00 verification	2.91	2.87	5.78
16	2200.00 computation	2.68	2.66	5.34
17	2200.00 verification	4.21	2.31	6.52
21	4100.00 computation	10.33	2.94	13.27
22	4100.00 verification	4.58	2.34	6.92
24	3200.00 computation	45.13	2.86	47.99
25	3200.00 verification	15.40	2.38	17.78
29	5100.00 computation	44.52	2.40	46.92
30	5100.00 verification	13.20	2.32	15.52
32	4200.00 computation	124.30	2.53	126.83
33	4200.00 verification	85.70	2.38	88.08
35	3300.00 computation	156.20	3.11	159.31
36	3300.00 verification	103.25	2.38	105.63
40	6100.00 computation	190.46	2.69	193.15
41	6100.00 verification	48.04	2.29	50.33
43	5200.00 computation	682.09	2.62	684.71
44	5200.00 verification	328.91	2.32	331.23
46	4300.00 computation	2963.39	2.37	2965.76
47	4300.00 verification	829.55	2.35	831.96
$\sum$	Total sum	5668.75	61.26	5730.01

Table 6.8: Time of computation and overhead for the medium-size experiment with supergroups.

6.7.4 Summary

By means of writing a simple DAG description script (such as the one shown in Figure 6.16), a workflow represented by a plain directed acyclic graph can be submitted to TrellisDAG. We showed that TrellisDAG can execute this workflow and satisfy inter-job dependencies in such a way that the resulting makespan is close to that of the model run.

We also showed how one can define supergroups in order to introduce a logical structure to the workflow; we pointed out that one has to weigh the convenience benefits from defining supergroups against the possible reduction of the degree of concurrency due to additional scheduling constraints introduced by the supergroups.

```
 1  #!/usr/bin/python
 2
 3  import sys
 4  import math
 5  import string
 6
 7  nPieces = int(sys.argv[1])
 8
 9  def isDependentPiece(piece1, piece2):
10      if (piece1[0] > piece2[0]):
11          return 1
12      return 0
13
14  #THE INTERFACE PART
15  Groups = [['Piece', 1], ['Rank', 6]]
16
17  def generateGroup(groupName, parentGroup):
18      result = []
19      if groupName == 'Piece':
20          for np in range(2, nPieces + 1):
21              result.append([np])
22      else:
23          np = parentGroup[0]
24          for b in range(1, np/2 + 1):
25              result.append([np - b, b, 0, 0, 0, 0])
26      return result
27
28  def getPrologueExecutables(level, group): return []
29  def getPrologueAttributes(level, group): return []
30  def getPrologueParameters(level, group): return []
31  def getEpilogueExecutables(level, group): return []
32  def getEpilogueAttributes(level, group): return []
33  def getEpiloguеParameters(level, group): return []
34
35  def getJobsExecutables(groupName, group):
36      if groupName == 'Rank':
37          return ["Bin/run.it", "Bin/ver.it"]
38      return []
39
40  def getJobsAttributes(groupName, group):
41      if groupName == 'Rank':
42          return [["release=" + "yes"], ["release=" + "no"]]
43      return []
44
45  def getJobsParameters(groupName, group):
46      if groupName == 'Rank':
47         return [string.join(map(str, group), " "),
48                 string.join(map(str, group), " ")]
49      return []
50
51  def isDependent(groupName, group1, group2, parentGroup):
52      if groupName == 'Rank':
53          return 0
54      return isDependentPiece(group1, group2)
```

Figure 6.29: The DAG description script for the all-kings computation with super-groups.

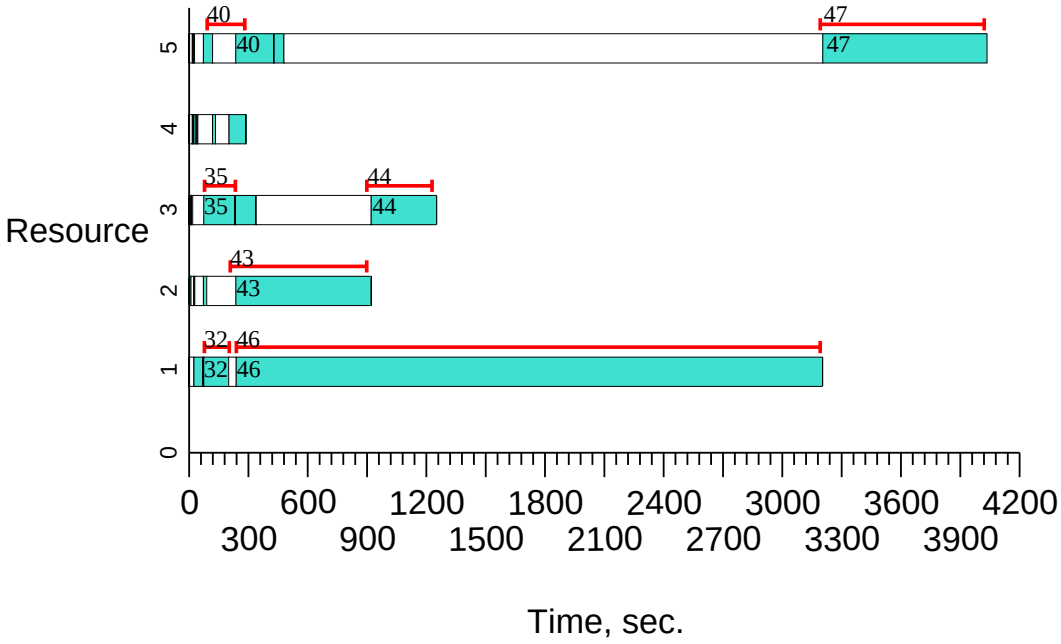

Figure 6.30: Resource utilization chart for the medium-size experiment with supergroups. The lines over bars represent the real run (see Figures 6.17, 6.18, and 6.19). See Table 6.8 for the definition of jobs.

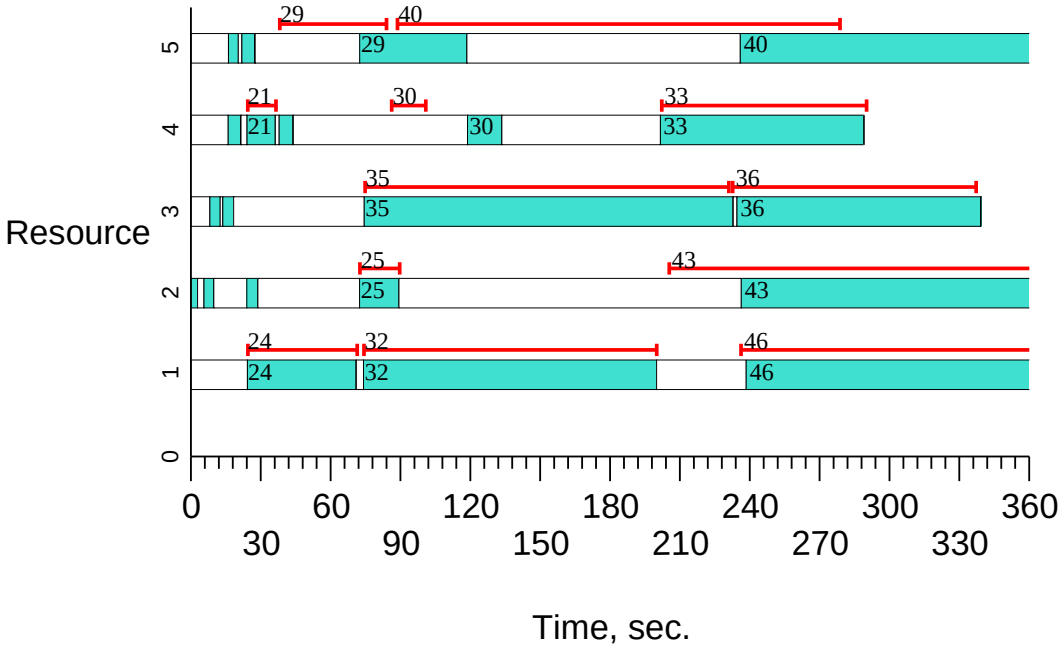

Figure 6.31: Resource utilization chart for the medium-size experiment with supergroups. This is a scaled version of the beginning part of the chart in Figure 6.30. The lines over bars represent the real run (see Figures 6.17, 6.18, and 6.19). See Table 6.8 for the definition of jobs.

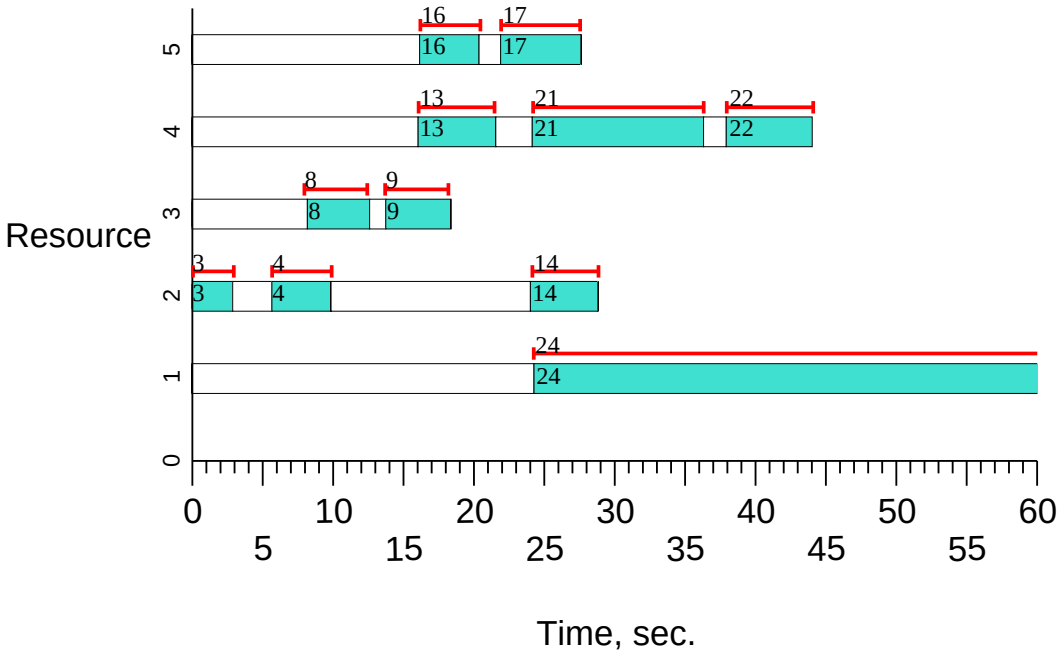

Figure 6.32: Resource utilization chart for the medium-size experiment with supergroups. This is a scaled version of the beginning part of the chart in Figure 6.31. The lines over bars represent the real run (see Figures 6.17, 6.18, and 6.19). See

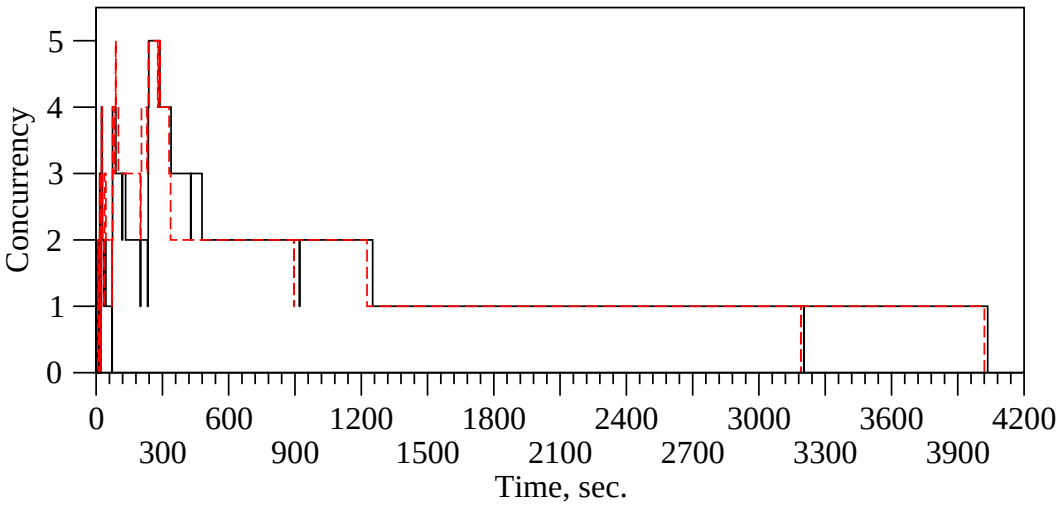

Figure 6.33: Degree of concurrency chart for the medium-size experiment with supergroups. The dashed graph represents the corresponding degrees of concurrency of the run without supergroups (see Figures 6.20, 6.21, and 6.22).

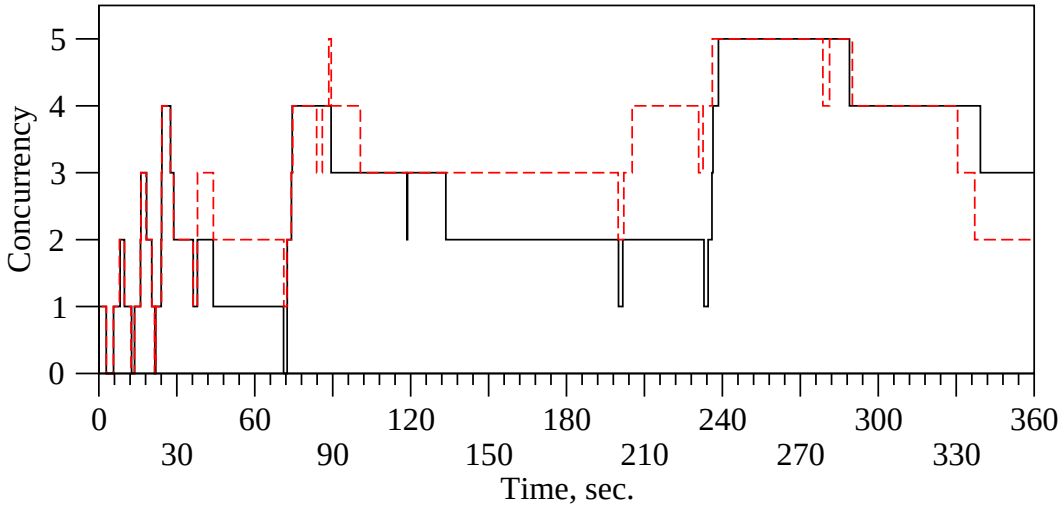

Figure 6.34: Degree of concurrency chart for the medium-size experiment with supergroups. This is a scaled version of the beginning part of the chart in Figure 6.33. The dashed graph represents the corresponding degrees of concurrency of the run without supergroups (see Figures 6.20, 6.21, and 6.22).

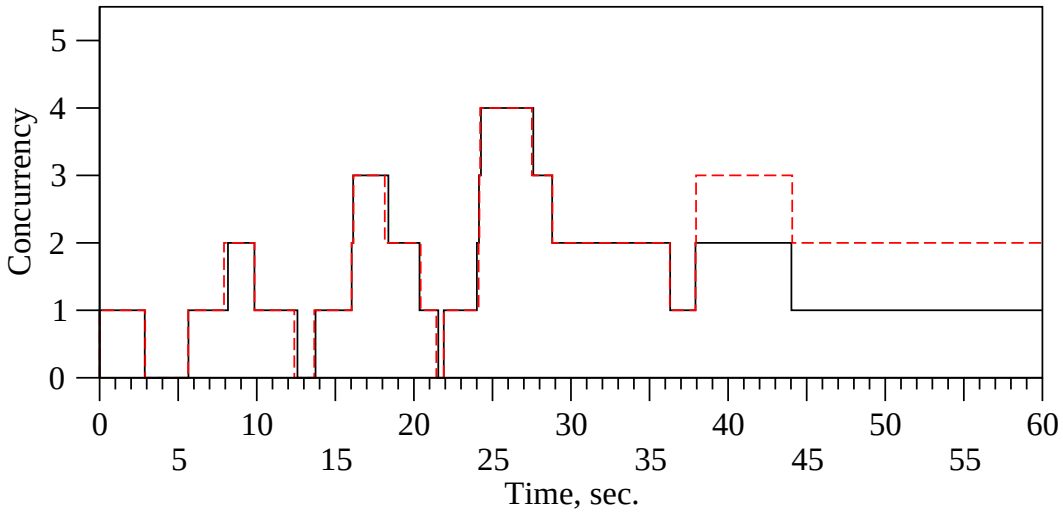

Figure 6.35: Degree of concurrency chart for the medium-size experiment with supergroups. This is a scaled version of the beginning part of the chart in Figure 6.34. The dashed graph represents the corresponding degrees of concurrency of the run without supergroups (see Figures 6.20, 6.21, and 6.22).

6.8 Large-size experiment: all 4 versus 3 pieces endgame databases

In this section, we are concerned with computation of all checkers endgame databases with 4 pieces of one color and 3 pieces of the other color on the board. We use the grouping capability of TrellisDAG and define 3 levels: `VS` (see Figure 3.3), `Slice` (see Figure 3.5) and `Rank` (see Figure 3.6).

The DAG description script for this computation is shown in Figure 6.36. We do not show the functions for the prologue and epilogue jobs that return empty lists.

```
 1  #!/usr/bin/python
 2
 3  import sys
 4  import math
 5  import string
 6
 7  nPieces = int(sys.argv[1])
 8
 9  def getDependentRank(rank):
10      return [rank[:4] + [rank[4] + 1, rank[5]],
11              rank[:4] + [rank[4], rank[5] + 1]]
12
13  def getDependentSlice(slice):
14      return [[slice[0] + 1, slice[1], slice[2] - 1, slice[3]],
15              [slice[0], slice[1] + 1, slice[2], slice[3] - 1]]
16
17  def getDependentVS(vs):
18      return [[vs[0] - 1, vs[1]], [vs[0], vs[1] - 1]]
19
20  #THE INTERFACE PART
21  Levels = [['VS', 2], ['Slice', 4], ['Rank', 6]]
22
23  def generateGroup(level, parentGroup):
24      result = []
25      if level == 'VS':
26          for np in range(2, nPieces + 1):
27              for b in range((np + 1)/2, np):
28                  result.append([b, np - b])
29      if level == 'Slice':
30          b = parentGroup[0]; w = parentGroup[1]
31          for bk in range(0, b + 1):
32              for wk in range(0, w + 1):
33                  bc = b - bk; wc = w - wk
34                  if bk + bc < wk + wc: continue
35                  if bk + bc == wk + wc and bc < wc: continue
36                  result.append([bk, wk, bc, wc])
37      if level == 'Rank':
38          brc = wrc = 1
39          if parentGroup[2] != 0: brc = 7
40          if parentGroup[3] != 0: wrc = 7
41          for br in range(0, brc):
42              for wr in range(0, wrc):
43                  result.append(parentGroup + [br, wr])
44      return result
45
46  def getJobsExecutables(level, group):
47      if level != 'Rank': return []
48      return ["Bin/run.it", "Bin/ver.it"]
49
50  def getJobsAttributes(level, group):
51      if level != 'Rank': return []
52      return [["release=yes"], ["release=no"]]
53
54  def getJobsParameters(level, group):
55      if level != 'Rank': return []
56      return [string.join(map(str, group), " "),
57              string.join(map(str, group), " ")]
58
59  def getDependent(level, group, parentGroup):
60      if level == 'VS': return getDependentVS(group)
61      if level == 'Slice': return getDependentSlice(group)
62      if level == 'Rank': return getDependentRank(group)
```

Figure 6.36: The DAG description script for the 4 pieces versus 3 pieces computation.

Experiment #	1	**2**	3	Minimal schedule
Makespan, sec.	32824.16	**32857.36**	32911.53	32362.38

Table 6.9: The makespans for the large-size experiment. The median run is shown in bold.

6.8.1 The experiment without data movement

Table 6.9 shows the makespans obtained in the runs. All of the presented charts correspond to the median run #2. The largest deviation from the makespan of the median run is $\frac{32911.53-32857.36}{32857.36} \approx 0.0016 = 0.16\%$.

The flow of computation is summarized with the resource utilization chart in Figure 6.37. We note that resources #1 and #16 were not used. The machine corresponding to the former resource did not have enough disk space left and the machine corresponding to the latter resource was used as a command-line server. Thus we used a total of $20 - 2 = 18$ processors and the same number of placeholders.

We point out a major difference between the workflow of this experiment and the workflow of the medium-size experiment. In the medium-size experiment, the shorter jobs tended to be at the beginning of the computation and the longer jobs tended to be closer to the end of the computation. There is no such pattern in the current experiment. Hence we do not provide the scaled versions of our charts. Note, however, that the first job of the current experiment is the same computation as the last job of the medium-size experiment, namely computing the slice `4300`.

The degree of concurrency chart is shown in Figure 6.39. We note that the maximal degree of concurrency achieved in the experiment is 18. Since this is exactly the number of placeholders that we used, the degree of concurrency could not possibly be higher and one may argue that there was a potential for a higher degree of concurrency if we used more processors. However, the total time when the degree of concurrency was 18 (or greater than 12 to that end) is very small and adding several more processors would not significantly affect the makespan.

We proceed to compare the results obtained in the current experiment with the minimal schedule that was obtained based on the pure computation/verification times in the median run of the experiment. The model run took 32362.38 sec-

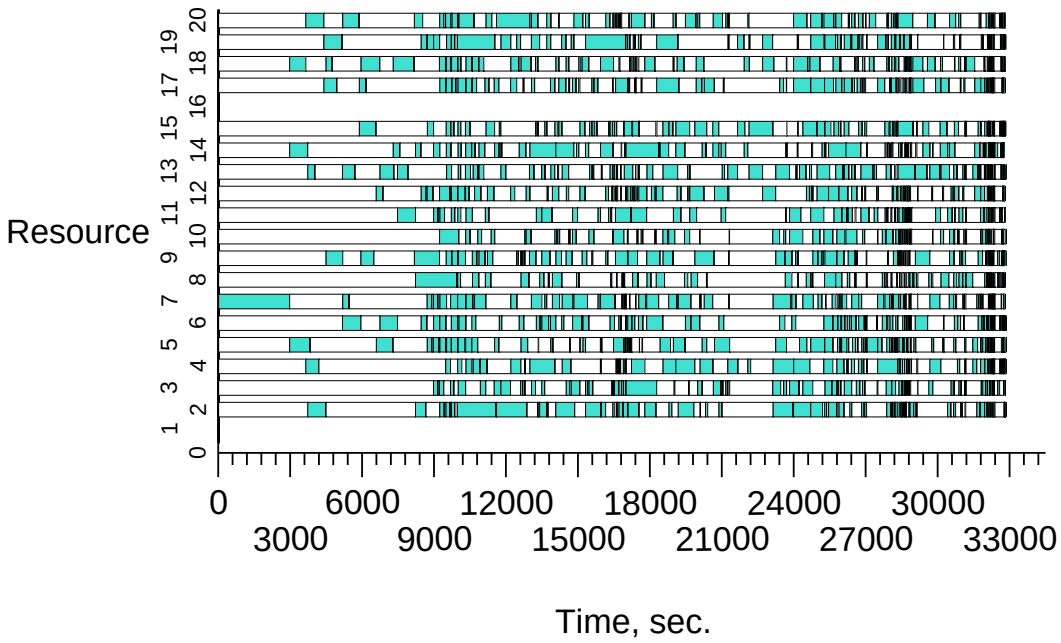

Figure 6.37: Resource utilization chart for the large-size experiment.

onds, which is only $\frac{32857.36-32362.38}{32857.36} \approx 0.0151 = 1.51\%$ faster than the makespan achieved in the real run.

The resource utilization chart in Figure 6.38 and the degree of concurrency chart in Figure 6.40 correspond to the minimal schedule. Note that the maximal degree of concurrency achieved by the minimal schedule is 19 and that the peak levels of concurrency were achieved approximately at the same time by the model computation and by the real run (compare Figure 6.39 and Figure 6.40). The degree of concurrency chart for the model run looks like a slightly condensed version of the chart for the real run.

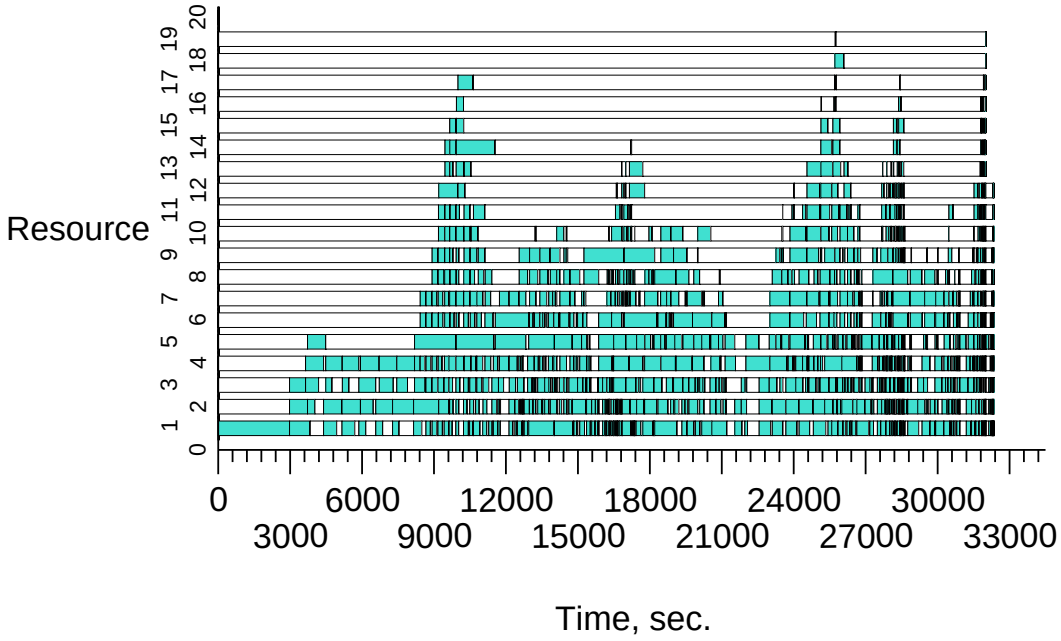

Figure 6.38: Resource utilization chart for the large-size experiment. Model run.

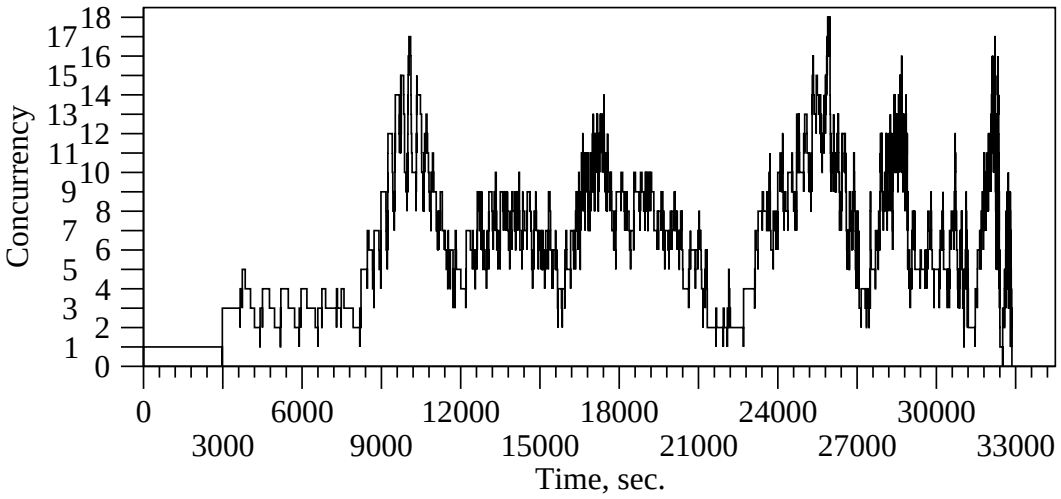

Figure 6.39: Degree of concurrency chart for the large-size experiment.

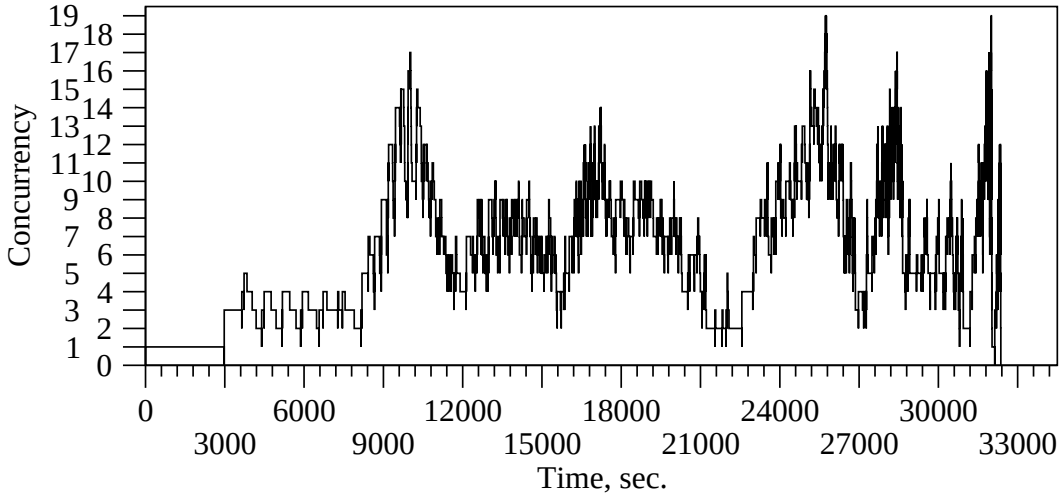

Figure 6.40: Degree of concurrency chart for the large-size experiment. Model run.

6.8.2 The experiment with data movement

We have demonstrated that TrellisDAG can efficiently perform the computation while respecting the application-specific workflow dependencies. However, thus far we did not concern ourselves with data movement. Since data movement is not integrated into the current version of TrellisDAG, we present the experiments with data movement separately in Appendix E. There we show how data movement can be performed within the framework of TrellisDAG and placeholder scheduling and explore the effect of using the affinity attribute on computing the 4 pieces versus 3 pieces checkers endgame databases. To summarize, we can draw two conclusions from our experiments with data movement:

1. Data movement can be performed within the framework of TrellisDAG and placeholder scheduling and

2. The benefits of using the affinity attribute due to the reduced number of copy operations is outweighted by the overheads due to the increased number of scheduling constraints. This conclusion is checkers-specific.

6.8.3 Summary

The large-size experiment is an example of using TrellisDAG for computing a complicated workflow with hundreds of jobs and inter-job dependencies. Writing a DAG description script of only several dozens of lines is enough to describe the whole workflow. As in the medium-size experiment, the makespan of the computation was close to that of the model run. At the end, we showed that the system can be used for real computations involving data movement.

6.9 Concluding remarks

We have shown how TrellisDAG can be effectively used to submit and execute three different workflows. Although all of these workflows correspond to different parts of the checkers computation, we believe that many prospective applications are represented by these workflows.

1. Multi-stage pipeline. Most of the applications of computational science fall in this category. We saw that such a pipeline can easily be submitted to the system without even using the DAG description script.

2. Plain DAG. Automatic building and testing of software could be an example of such a workflow.

3. Large DAG with a structure imposed by supergroups. This is the kind of workflow that we are dealing with when computing the checkers endgame databases – our motivating application.

In all cases, writing the DAG description scripts for the workflows turned out to be a straightforward task. The lessons gained from the execution of our experiments can be summarized as follows:

1. The makespans achieved by TrellisDAG are consistently close to the makespans achieved by the minimal schedule. We conclude that the system effectively utilizes the opportunities for concurrent execution,

2. Using the supergroups capability of the system may involve a trade-off between convenience of monitoring and efficiency of computation, and

3. Data movement can be incorporated into a computation under TrellisDAG. Using the affinity attribute involves a trade-off between the time savings due to making fewer data movements and potential reduction of the degree of concurrency due to extra scheduling constraints.

Chapter 7

Future Work and Concluding Remarks

In the previous chapters, we described the design and implementation of the TrellisDAG system and put this work in perspective with previous work in the field of high-performance computing; furthermore, we evaluated the features of TrellisDAG through experimental studies in Chapter 6. In this chapter, we provide directions for future work and make concluding remarks for the thesis.

7.1 Future work

In this section, we briefly summarize some possible directions for future efforts to improve the TrellisDAG system.

1. **Separation of convenience issues from scheduling issues.** We saw examples when defining supergroups was convenient from the monitoring point of view, but degrading from the scheduling point of view. This problem can be resolved by enabling the user to define supergroups that are only used for monitoring and administering, but do not add any implicit workflow dependencies.

2. **Implementing known scheduling policies.** Using the mechanisms for scheduling introduced in this thesis, we can implement several known policies for DAG scheduling, e.g. shortest/largest job first [17].

3. **Group-level scheduling.** Since we define dependencies between groups rather than between jobs, it is natural to make scheduling decisions in group terms. Moreover, since we have groups at several levels, we can make scheduling decisions at several levels. For example, we could define partitions of resources and assign groups at the highest level to partitions. Later, we could schedule groups at lower levels within their respective partitions. The partitions could correspond to administrative domains and match to groups according to some properties of such domains (i.e. timing constraints). In order to take most benefit from group-level scheduling, TrellisDAG would be augmented with several features:

   (a) Groups have to be stored explicitly in the jobs database,

   (b) Groups at all levels have to support attributes, and

   (c) The interface of the command-line server has to be modified to support scheduling all jobs of a group at once.

7.2 Concluding remarks

We have designed and implemented TrellisDAG – a system that combines the use of placeholder scheduling and a subsystem for describing workflows to provide the necessary mechanisms for computing non-trivial workloads with inter-job dependencies.

The design of TrellisDAG allows it to handle many prospective applications and we used computing the checkers end-game databases as a motivation and proof-of-concept. The checkers application gives rise to a rich variety of workflows – from simple to rather complicated – and, effectively, lets us show how generic the TrellisDAG's design is. We used three such workflows to establish that TrellisDAG provides a convenient tool for describing complicated workflows, executes the workflow without violating any inter-job dependencies, and achieves a schedule that is close to the minimal schedule (i.e. makes good utilization of the application-specific opportunities for concurrent execution of jobs).

TrellisDAG provides a modular architecture for describing scheduling policies. We believe that the combination of this mechanism with the mechanism of jobs' and groups' attributes creates a generic framework in which static and dynamic (e.g. learning) scheduling policies can be implemented (see Section 7.1).

Bibliography

[1] D. Abramson, R. Sosic, J. Giddy, and B. Hall. Nimrod: A tool for performing parametised simulations using distributed workstations. In *The 4th IEEE Symposium on High Performance Distributed Computing, Virginia*, August 1995.

[2] B. A. Kingsbury. The Network Queueing System (NQS). http://ory.ph.biu.ac.il/manuals/NQS/, 1992.

[3] Condor. http://www.cs.wisc.edu/condor/.

[4] CondorG. http://www.cs.wisc.edu/condor/condorg.

[5] DAGMan Metascheduler. http://lxmi.mi.infn.it/condor/manual/2_10Interjob_Dependencies.html.

[6] DataTAG home page. http://datatag.web.cern.ch/datatag/.

[7] DOE Science Grid. http://doesciencegrid.org/.

[8] D. G. Feitelson, L. Rudolph, U. Schwiegelshohn, K. C. Sevcik, and P. Wong. Theory and Practice in Parallel Job Scheduling. In D. G. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, volume 1291 of *Lecture Notes in Computer Science*, pages 1–34. Springer-Verlag, 1997.

[9] I. Foster. What is the grid? A three point checklist. *Daily News and Information for the Global Grid Community*, 1(6), July 2002. Available at http://www.gridtoday.com/02/0722/100136.html.

[10] I. Foster and C. Kesselman. The Globus project : A status report. *Proc. IPPS/SPDP '98 Heterogeneous Computing Workshop*, pages 4–18, 1998.

[11] I. Foster, C. Kesselman, J. M. Nick, and S. Tecke. The Physiology of the Grid: An Open Grid Services Architecture for Distributed System Integration, June 2002.

[12] Global Grid Forum. http://www.gridforum.org/.

[13] Globus. http://www.globus.org/.

[14] Grid Physics Network. http://www.griphyn.org.

[15] HPCC Grand Challenges. http://www.nhse.org/grand_challenge.html.

[16] iVDGL - International Virtual Data Grid Laboratory. http://www.ivdgl.org.

[17] Y. Kwok and I. Ahmad. Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors. *ACM Computing Surveys*, 31(4):406–471, December 1999.

[18] R. Lake and J. Schaeffer. Solving the Game of Checkers. In Richard J. Nowakowski, editor, *Games of No Chance*, pages 119–133. Cambridge University Press, 1996.

[19] R. Lake, J. Schaeffer, and P. Lu. Solving Large Retrograde Analysis Problems Using a Network of Workstations. *Advances in Computer Chess*, VII:135–162, 1994.

[20] Legion. http://www.cs.virginia.edu/~legion/.

[21] M. Lewis and A. Grimshaw. The core Legion object model. In *Proceedings of the Fifth IEEE International Symposium on High Performance Distributed Computing*, August 1996.

[22] M. Lewis, W. A. Wulf, and the whole Legion team. Legion – a view from 50,000 feet. In *Proceedings of the Fifth IEEE International Symposium on High Performance Distributed Computing*, August 1996.

[23] M. Livny, J. Basney, R. Raman, and T. Tannenbaum. Mechanisms for high throughput computing. *SPEEDUP*, 11, 1997.

[24] Load Sharing Facility (LSF). http://www.platform.com/.

[25] LoadLeveler. http://www.mhpcc.edu/training/workshop/loadleveler/MAIN.html.

[26] P. Lu. Parallel Search of Narrow Game Trees. Master's thesis, Dept. of Computing Science, University of Alberta, Edmonton, Alberta, Canada, 1993.

[27] NASA Information Power Grid (IPG). http://www.ipg.nasa.gov/.

[28] P. Gburzynski. Operating Systems Concepts. http://www.cs.ualberta.ca/~pawel/COURSES/379/.

[29] Particle Physics Data Grid. http://www.ppdg.net.

[30] C. Pinchak. Placeholder Scheduling for Overlay Metacomputers. Master's thesis, Dept. of Computing Science, University of Alberta, Edmonton, Alberta, Canada, 2002.

[31] C. Pinchak and P. Lu. Adaptive Job Scheduling in Overlay Metacomputers. in preparation.

[32] C. Pinchak, P. Lu, and M. Goldenberg. Practical Heterogeneous Placeholder Scheduling in Overlay Metacomputers: Early Experiences. In *8th Workshop on Job Scheduling Strategies for Parallel Processing*, Edinburgh, Scotland, U.K., July 24 2002.

[33] C. Pinchak, P. Lu, J. Schaeffer, and Mark Goldenberg. The canadian internetworked scientific supercomputer. In *17th Annual International Symposium on High Performance Computing Systems and Applications (HPCS)*, Sherbrooke, Quebec, Canada, May 2003.

[34] Portable Batch System. http://www.openpbs.org/.

[35] PostgreSQL Database Management System. http://www.postgresql.org/.

[36] J. Siegel and P. Lu. User-Level Remote Data Access in Overlay Metacomputers. In *Proceedings of the 4th IEEE International Conference on Cluster Computing*, September 2002.

[37] A. Silberschatz and P. B. Galvin. *Operating System Concepts*. Addison-Wesley, 1998.

[38] Sun Grid Engine. http://www.sun.com/software/gridware/sge.html.

[39] TeraGrid. http://www.teragrid.org/.

[40] The DataGrid Project. http://eu-datagrid.web.cern.ch/eu-datagrid/.

[41] The Nimrod/G project. http://www.csse.monash.edu.au/~sgaric/nimrod/.

[42] The Trellis Project. http://www.cs.ualberta.ca/~paullu/Trellis/.

[43] S. Zhou, X. Zheng, J. Wang, , and P. Delisle. Utopia: A load sharing facility for large heterogeneous distributed computer systems. In *Software Practice and Experience,* December 1993.

Appendix A

The Placeholder for the Checkers Computation

```
 1  #!/bin/sh
 2
 3  # (c) Mark Goldenberg 2002
 4
 5  # SGE Placeholder template
 6
 7  # request Bourne shell as shell for job
 8  #$ -S /bin/sh
 9
10  exec 2>&1
11
12  #the path to the directory used at this machine, this placeholder is there
13  LOCAL_PATH=/usr/scratch/checkers
14  cd $LOCAL_PATH
15  . ./common.sh # listed in Figure A.2.
16
17  function submitPlaceholder {
18      if [ $SUBMIT != $host ]; then
19          $SSH -l $user $SUBMIT $SGE_BIN/qsub $SUBMIT_PATH/$placeholder
20      else
21          $SGE_BIN/qsub $LOCAL_PATH/$placeholder
22      fi
23  }
24
25  function runApplication {
26      echo Running $@
27      output=‘csh -c "$@"‘
28      result=$?
29
30      if [ $result -eq 255 ]; then
31          echo "Failed $@; return value: $result"
32          submitPlaceholder
33          exit 1
34      fi
35  }
36
37  #take complete file name as parameter and set $dir to be the path part
38  function dirName {
39      dir=‘/usr/bin/dirname $1‘
40  }
41
42  #this does not support proxy; for thesis experiment only
43  #uses NFS
44  function sendData {
45      cd $LOCAL_PATH
```

```
46
47      #remove uncompressed databases
48      rm -f ‘find ? | egrep -v v1‘ 2>/dev/null
49
50      files=‘find ?‘
51
52      if [ -z "$files" ]; then
53          echo "No data to send!"
54          return
55      fi
56      md5sum $files > md5.res 2>/dev/null
57
58      files=‘cat md5.res | colrm 1 34‘
59      if [ -z "$files" ]; then
60          echo "No data to send!"
61          return
62      fi
63
64
65      #copy the files to jasper-reserve/
66      data_dir=$SERVER_PATH/jasper-reserve
67      md5res=1
68      while [ $md5res -ne 0 ]; do
69          /usr/bin/rcp -r $LOCAL_PATH/? $user@brule:$data_dir/
70          echo $data_dir
71          md5sum --check $LOCAL_PATH/md5.res #>&/dev/null
72          md5res="$?"
73      done
74
75      #create links in jasper-??/
76      $SSH $user@brule "cd $SERVER_PATH; ./links.sh $host "$files""
77
78      cd $LOCAL_PATH
79
80      for name in $files; do
81          toBuild=‘echo $name | grep idx | colrm 1 11 | colrm 8‘
82          Bin/build $toBuild >& /dev/null
83      done
84
85      rm -rf ?
86  }
87
88  #this does not support proxy; for thesis experiment only
89  #uses NFS
90  function getData {
91      files=‘$SSH $user@brule "cd $SERVER_PATH/$host; find ?"‘
92
93      if [ -z "$files" ]; then
94          echo "No data to get!"
95          return
96      fi
97
98      $SSH $user@brule "cd $SERVER_PATH/$host; md5sum "$files" > md5.res"
99
100     cd $SERVER_PATH/$host
101     files=‘cat md5.res | colrm 1 34‘
102     if [ -z "$files" ]; then
103         echo "No data to get!"
104         return
105     fi
106
107     cd $LOCAL_PATH
108
109     md5res=1
110     while [ $md5res -ne 0 ]; do
111         #copy everything; later we still use only the ones in $files
112         /usr/bin/rcp -r $user@brule:$SERVER_PATH/$host/* .
113         md5sum --check $SERVER_PATH/$host/md5.res #>& /dev/null
```

```
114         md5res="$?"
115     done
116
117     for name in $files; do
118         rm -f $SERVER_PATH/$host/$name
119         toBuild=‘echo $name | grep idx | colrm 1 11 | colrm 8‘
120         Bin/build $toBuild >& /dev/null
121     done
122
123     rm -rf ?
124 }
125
126 setTimer 0
127 dateMessage "STARTING_ON_$host"
128
129 #make sure that the server knows about this placeholder and
130 #make sure that this placeholder does not exceed the time limit
131 #./nanny.sh $JOB_ID &
132
133 setTimer 1
134 dateMessage "GETTING_NEXT_JOB"
135 retry invokeMQService mqnextjob.py sched=SGE "sched_id=$JOB_ID" \
          submit_host=$SUBMIT host=$host
136 timeMessage 1 "GOT_NEXT_JOB_($output)"
137
138 id=$output
139
140 if [ $id -lt 0 ]; then
141     rm -f $HOME/$placeholder.?$JOB_ID
142     exit 111
143 fi
144
145 if [ $id -eq 0 ]; then
146     sleep 5
147     submitPlaceholder
148     rm -f $HOME/$placeholder.?$JOB_ID
149     exit 0
150 fi
151
152 setTimer 1
153 dateMessage "GETTING_COMMAND_LINE"
154 retry invokeMQService mqgetjobcommand.py "id=$id"
155 OPTIONS="$output"
156 timeMessage 1 "GOT_COMMAND_LINE"
157
158 setTimer 1
159 dateMessage "GETTING_AFFINITY_ATTRIBUTE"
160 retry invokeMQService mqgetjobattribute.py "id=$id" attribute=affinity
161 affinity="$output"
162 timeMessage 1 "GOT_AFFINITY_ATTRIBUTE"
163
164 if [ "$affinity" != "yes" ]; then
165     setTimer 1
166     dateMessage "GETTING_DATA"
167     getData
168     timeMessage 1 "DATA_HAS_BEEN_OBTAINED"
169 fi
170 cd $LOCAL_PATH
171
172 setTimer 1
173 echo $OPTIONS
174 runApplication "$OPTIONS"
175 timeMessage 1 "DONE_WITH_computation/verification"
176 seqTime=${diff[1]}
177
178 setTimer 1
179 dateMessage "SENDING_DATA"
180 sendData
```

```
timeMessage 1 "DATA_HAS_BEEN_SENT"
cd $LOCAL_PATH

setTimer 1
dateMessage "CALLING_mqdonejob.py"
retry invokeMQService mqdonejob.py "id=$id"
timeMessage 1 "mqdonejob.py_IS_SUCCESSFUL"

setTimer 1
submitPlaceholder
rm -f $HOME/$placeholder.e$JOB_ID
timeMessage 1 "removed stderr file and resubmitted"

timeMessage 0 "FINISHED_SUCCESSFULLY!"
echo GANTT:${id}_${host}_${start[0]}_${diff[0]}_$seqTime
```

Figure A.1: The placeholder for the checkers computation.

```

#the host to use for submission
SUBMIT=‘hostname‘

#the path to the placeholder at $SUBMIT,
#empty if same machine (i.e. if SUBMIT==‘hostname‘)
SUBMIT_PATH=

HOME="/usr/brule18/grad/goldenbe"

#where the qsub program is located
SGE_BIN="$HOME/SGE/bin/glinux/"

#name of the placeholder at $SUBMIT/$SUBMIT_PATH
placeholder=template.sh

#the path to the directory used at this machine, this placeholder is there
LOCAL_PATH=/usr/scratch/checkers

#the host with the command-line server
SERVER=jasper-15

#the path at the server
SERVER_PATH=$HOME/build/Exp

#submit host
SUBMIT=‘hostname‘

#the path to MQ services on the server
MQ_PATH=$HOME/build/Bin/server

#the public key to be used for connections
KEY=$HOME/.ssh/temp

SSH="ssh -i $KEY"
SCP="scp -i $KEY -r"
host=‘hostname‘
user=‘whoami‘

#job ID initialized to be empty
id=

function mytime {
    /usr/brule18/grad/goldenbe/build/QCheckers/mytime
}
```

```
46
47  function retry {
48      try=0
49      result1=1
50      result2=0
51      while [ $result1 -ne 0 -o $result2 -ne 0 ]; do
52          if [ $try -gt 0 ]; then
53              dateMessage "Operation failed; retrying..."
54          fi
55          "$@"
56          try=$[$try + 1]
57      done
58  }
59
60  function invokeMQService {
61     if [ -z $PROXY ]; then
62          output=‘$SSH $SERVER -l $user "$MQ_PATH/$*"‘
63          result1=$?
64      else
65          output=‘$SSH $PROXY -l $user $SSH $SERVER -l $user "$MQ_PATH/$*"‘
66          result1=$?
67      fi
68  }
69
70  function setTimer {
71      start[$1]=‘mytime‘
72  }
73
74  #$1 - timer number
75  #$2 - message, the same message goes to the database as status
76  function timeMessage {
77      timerUpdate $1
78      echo TIMER MESSAGE: $2 in ${diff[$1]}\sec. ‘date "+at %H:%M:%S on %y/%m/%d"‘ \
            \(‘mytime‘\)
79      if [ ! -z $id ]; then
80          :
81          #invokeMQService mqstatus.py "id=$id" "status=$2"
82      fi
83  }
84
85  #$1 - timer number
86  function timerUpdate {
87      end[$1]=‘mytime‘
88      diff[$1]=‘python -c "print ’%.2f’ % (${end[$1]} - ${start[$1]})"‘
89  }
90
91  #$1 - message, the same message goes to the database as status
92  function dateMessage {
93      echo MESSAGE: "$1" ‘date "+at %H:%M:%S on %y/%m/%d"‘ \(‘mytime‘\)
94
95      if [ ! -z $id ]; then
96          :
97          #invokeMQService mqstatus.py "id=$id" "status=$1"
98      fi
99  }
```

Figure A.2: `common.sh`: Routines used in the placeholder and elsewhere.

Appendix B

The Placeholder Nanny

```
 1  #!/bin/bash
 2
 3  . common.sh
 4
 5  TOTAL_TIME_LIMIT=7200
 6  DELAY=10
 7  sched_id=$1
 8
 9  exec >$HOME/$placeholder.o$sched_id
10
11  echo "The nanny started"
12  setTimer 99
13
14  while [ 1 ]; do
15      retry invokeMQService mqsignal.py \
16          sched=SGE sched_id=$sched_id submit_host=$SUBMIT
17      timerUpdate 99
18      #converting the real time to an integer
19      diff[99]=‘python -c "print int(${diff[99]})"‘
20      if [ ${diff[99]} -gt $TOTAL_TIME_LIMIT ]; then
21          echo "Over time limit"
22          break
23      fi
24      sleep $DELAY
25  done
26
27  echo "Killing my placeholder" 1>&2
28  $SSH -l $user $SUBMIT "$SGE_BIN/qdel $sched_id"
29
```

Figure B.1: The placeholder nanny. See Figure A.2 for the listing of `common.sh`

Appendix C

Rules of Checkers in Brief

Rules of Checkers In Brief[1]

Standard checkers notation assigns each square with a number, as in Figure A.1. Moves are given as from-square/to-square pairs. Black initially occupies squares 1 to 12 inclusive and White occupies squares 21 to 32 inclusive. If an opponent's checker is on an adjacent square, and there is an empty square behind it on the same diagonal, it can be captured (i.e. jumped) and removed from the board. In checkers, captures must be taken. Multiple captures are allowed and if more than one capture is possible, the player can choose. Checkers can only move and capture forwards. When a checker reaches the opponent's back rank (i.e. a White checker reaches one of 1 to 4, a Black checker reaches one of 29 to 32), it is promoted to a king. Kings can move and capture both forwards and backwards. A player wins if the opponent has no more pieces on the board, or if the opponent has no more legal moves. Black moves first.

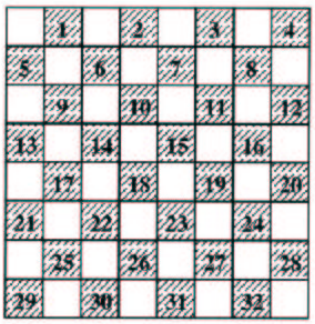

Figure A.1 — Checkers Board Notation

[1] Adapted from [STL93]

Figure C.1: This figure provides the brief description of the rules of checkers as it appeared in Paul Lu's Masters thesis [26].

Appendix D

Code Listing for `mqnextjob` and `mqdonejob` Services of the Command-Line Server

```
 1  #!/usr/bin/python
 2
 3  # Written by Mark Goldenberg, April 2002
 4
 5  import sys
 6  import pg
 7  import os
 8  import string
 9  import time
10  from Bin import common
11  from Bin import schema
12  from Bin.common import myquery
13
14  def setStartTime(j_id):
15      query_str = "SELECT * FROM Jobs WHERE j_id = " + str(j_id)
16      query_result = myquery(query_str)
17      jobTuple = (query_result.dictresult())[0]
18      attrs = common.getDictFromString(jobTuple['attrs'])
19      attrs["lastStart"] = str(time.time())
20      query_str = "UPDATE Jobs SET attrs =" + \
21                  "'" + common.getStringFromDict(attrs) + "'" + \
22                  " WHERE j_id = " + str(j_id)
23      query_result = myquery(query_str)
24
25  def setEndTime(j_id):
26      query_str = "SELECT * FROM Jobs WHERE j_id = " + str(j_id)
27      query_result = myquery(query_str)
28      jobTuple = (query_result.dictresult())[0]
29      attrs = common.getDictFromString(jobTuple['attrs'])
30      attrs["lastCompletion"] = str(time.time())
31      query_str = "UPDATE Jobs SET attrs =" + \
32                  "'" + common.getStringFromDict(attrs) + "'" + \
33                  " WHERE j_id = " + str(j_id)
34      query_result = myquery(query_str)
35
36  def setHost(j_id, host):
37      query_str = "SELECT * FROM Jobs WHERE j_id = " + str(j_id)
38      query_result = myquery(query_str)
39      jobTuple = (query_result.dictresult())[0]
40      attrs = common.getDictFromString(jobTuple['attrs'])
41      attrs["lastHost"] = host
42      query_str = "UPDATE Jobs SET attrs =" + \
```

```
43                 "'" + common.getStringFromDict(attrs) + "'" + \
44                 " WHERE j_id = " + str(j_id)
45     query_result = myquery(query_str)
46 
47 def findFirstJob(tar_id):
48     query_str = "SELECT MIN(j_id) FROM Jobs WHERE tar_id = " + \
49                 str(tar_id) + " AND status = " + str(schema.MQ_NOT_STARTED)
50     query_result = myquery(query_str)
51     # if empty, min is 0, so no need for try block
52     return query_result.dictresult()[0]['min']
53 
54 def noJobs():
55     query_str = "SELECT * FROM Jobs WHERE status <> " + \
56                 str(schema.MQ_DONE)
57 
58     query_result = myquery(query_str)
59     if query_result.ntuples() == 0:
60         print -1
61         common.closeConnection()
62         sys.exit(0)
63     else:
64         print 0
65         common.closeConnection()
66         sys.exit(0)
67 
68 def restoreCache():
69     tar_ids = common.readTable("Targets", ['tar_id', 'status'])
70     #Have to get all with NOT_STARTED and SUBGROUPS_IN_PROGRESS
71     #That is not DONE or DISABLED or RUNNING
72     tar_ids = common.getTuplesGivenNotField(tar_ids, 1, schema.MQ_DONE)
73     tar_ids = common.getTuplesGivenNotField(tar_ids, 1, schema.MQ_DISABLED)
74     tar_ids = common.getTuplesGivenNotField(tar_ids, 1, schema.MQ_RUNNING)
75     tar_ids = common.getProjectionList(tar_ids, 0)
76     tar_ids.sort()
77 
78     dep_ids = common.readTable("Before", ['dep_id', 'status'])
79     dep_ids = common.getTuplesGivenField(dep_ids, 1, schema.MQ_ACTIVE)
80     dep_ids = common.getProjectionList(dep_ids, 0)
81     dep_ids.sort()
82 
83     goodTargets = []
84     tarCounter = 0
85     depCounter = 0
86     while tarCounter < len(tar_ids) and depCounter < len(dep_ids):
87         if tar_ids[tarCounter] < dep_ids[depCounter]:
88             while tar_ids[tarCounter] < dep_ids[depCounter]:
89                 goodTargets.append(tar_ids[tarCounter])
90                 tarCounter = tarCounter + 1
91                 if tarCounter >= len(tar_ids):
92                     break
93         elif tar_ids[tarCounter] > dep_ids[depCounter]:
94             while tar_ids[tarCounter] > dep_ids[depCounter]:
95                 depCounter = depCounter + 1
96                 if depCounter >= len(dep_ids):
97                     break
98         else:
99             tarValue = tar_ids[tarCounter]
100             while tar_ids[tarCounter] == tarValue:
101                 tarCounter = tarCounter + 1
102                 if tarCounter >= len(tar_ids):
103                     break
104             depValue = dep_ids[depCounter]
105             while dep_ids[depCounter] == depValue:
106                 depCounter = depCounter + 1
107                 if depCounter >= len(dep_ids):
108                     break
109 
110     if depCounter == len(dep_ids):
```

```
111         for j in  range(tarCounter, len(tar_ids)):
112             goodTargets.append(tar_ids[j])
113 
114     return goodTargets
115 
116 def newTargets(tar_id):
117     result = []
118 
119     query_str = "SELECT dep_id FROM BEFORE WHERE pre_id = " + str(tar_id)
120     query_result = myquery(query_str)
121     targets = query_result.dictresult()
122     for target in targets:
123         target = target['dep_id']
124         query_str = "SELECT pre_id FROM BEFORE WHERE dep_id = " + \
125                     str(target) + " AND " + \
126                     "status = " + str(schema.MQ_ACTIVE)
127         query_result = myquery(query_str)
128         #print target
129         #print query_result
130         if (query_result.ntuples() == 0):
131             query_str = "SELECT status FROM Targets WHERE tar_id = " + \
132                         str(target)
133             query_result = myquery(query_str)
134             if (query_result.dictresult()[0]['status'] ==
135                 schema.MQ_NOT_STARTED or
136                 query_result.dictresult()[0]['status'] ==
137                 schema.MQ_SUBGROUPS_IN_PROGRESS):
138                 result.append(target)
139     #print "new targets are ", result
140     return result
141 
142 def addToCache(goodTargets):
143     for target_id in goodTargets:
144         insert_id = findFirstJob(target_id)
145         # because of insertion in mqdonejob.py, need a check
146         # not to instroduce dublicates
147         query_str = "SELECT j_id FROM Cache WHERE j_id = " + str(insert_id)
148         query_result = myquery(query_str)
149         if query_result.ntuples() == 0:
150             query_str = "INSERT INTO Cache VALUES(" \
151                         + str(insert_id) + ", " + "''" + ")"
152             query_result = myquery(query_str)
153             #print insert_id, "is put in!"
154 
155 def done(j_id, host):
156     query_str = "SELECT * FROM Jobs WHERE j_id = " + \
157                 str(j_id)
158     query_result = myquery(query_str)
159     jobTuple = (query_result.dictresult())[0]
160     target_id = jobTuple['tar_id']
161     attrs = common.getDictFromString(jobTuple['attrs'])
162     target = common.getField("Targets", "tar_id", target_id,
163                              "tar_name")
164     command = jobTuple['comm_line']
165 
166     #delete from Cache (it could still be there)
167     query_str = "DELETE FROM Cache WHERE j_id = " + str(j_id)
168     query_result = myquery(query_str)
169 
170     #delete from Running
171     query_str = "DELETE FROM Running WHERE j_id = " + str(j_id)
172     query_result = myquery(query_str)
173 
174     # Update the status field in Jobs, Before
175     query_str = "UPDATE Jobs SET status = " + str(schema.MQ_DONE) + " " + \
176                 "WHERE j_id = " + str(j_id)
177     query_result = myquery(query_str)
178 
```

```
179     #check if this was the last job for the target
180     query_str = "SELECT * FROM Jobs WHERE tar_id = " + str(target_id) + \
181                 " AND status <> " + str(schema.MQ_DONE)
182     query_result = myquery(query_str)
183     isLast = (query_result.ntuples() == 0)
184
185     # Update the status field in Targets and Before Table
186     query_str1 = "UPDATE Targets SET status = " + \
187                  str(schema.MQ_SUBGROUPS_IN_PROGRESS) + \
188                  " " + "WHERE tar_id = " + str(target_id)
189     query_str2 = "UPDATE Targets SET attr_status = " + \
190                  str(schema.MQ_ATTR_NONE) + " " + \
191                  "WHERE tar_id = " + str(target_id)
192     release = "no"
193
194     try:
195         release = attrs["release"]
196     except:
197         pass
198
199     if isLast or release == "yes":
200         query_str = "UPDATE Before SET status = " + \
201                     str(schema.MQ_NOT_ACTIVE) + " " + \
202                     "WHERE pre_id = " + str(target_id)
203         query_result = myquery(query_str)
204
205         if isLast:
206             query_str1 = "UPDATE Targets SET status = " + \
207                          str(schema.MQ_DONE) + " " + \
208                          "WHERE tar_id = " + str(target_id)
209         if release == "yes" and not isLast:
210             query_str2 = "UPDATE Targets SET attr_status = " + \
211                          str(schema.MQ_ATTR_RELEASED) + " " + \
212                          "WHERE tar_id = " + str(target_id)
213
214     myquery(query_str1)
215     if query_str2 != "":
216         myquery(query_str2)
217
218     #fill the Cache table with jobs that became available
219     if not isLast:
220         query_str_aff = "SELECT * FROM Jobs WHERE j_id = " + str(j_id + 1)
221         query_result_aff = myquery(query_str_aff)
222         jobTuple = (query_result_aff.dictresult())[0]
223         attrs = common.getDictFromString(jobTuple['attrs'])
224         affinity = "no"
225         try:
226             affinity = attrs["affinity"]
227         except:
228             pass
229         if affinity <> "yes":
230             host = ""
231         query_str = "INSERT INTO Cache VALUES(" + \
232                     str(j_id + 1) + ", " + \
233                     "'" + host + "'" + ")"
234         myquery(query_str)
235         #print j_id + 1, "is put in!"
236
237     if isLast or release == "yes":
238         #if isLast:
239         #    print "Last!"
240         #else:
241         #    print "Release!"
242         targets = newTargets(target_id)
243         addToCache(targets)
244
245     return command
246
```

```
247  def next(stamp):
248     #delete the entry from Cache
249     #query_str = "DELETE FROM Cache WHERE j_id = " + stamp
250     #query_result = myquery(query_str)
251
252     # get the job tuple
253     query_str = "SELECT * FROM Jobs WHERE j_id = " + stamp
254     query_result = myquery(query_str)
255     jobTuple = query_result.dictresult()[0]
256     command = jobTuple['comm_line']
257     attrs = jobTuple['attrs']
258     target_id = jobTuple['tar_id']
259     target = common.getField("Targets",
260                              'tar_id', target_id, 'tar_name')
261
262     #change status field in Jobs, Targets
263     query_str = "UPDATE Jobs SET status = " + str(schema.MQ_RUNNING) + " " + \
264                 "WHERE j_id = " + stamp
265     query_result = myquery(query_str)
266
267     query_str = "UPDATE Targets SET status = " + str(schema.MQ_RUNNING) + " " + \
268                 "WHERE tar_id = " + str(target_id)
269     query_result = myquery(query_str)
270
271     #update parents' status fields
272     while 1:
273         level_id = common.getField("Targets", "tar_name", target,
274                                    "level_id")
275         if level_id == 0:
276             break
277
278         parent = schema.getParent(target)
279         status = common.getField("Targets", "tar_name", parent,
280                                  "status")
281
282         if status == schema.MQ_NOT_STARTED:
283             common.updateField("Targets", "tar_name", parent,
284                                "status", schema.MQ_SUBGROUPS_IN_PROGRESS)
285             target = parent
286         else:
287             break
288
289     #insert a tuple into Running
290     started = str(time.time())
291
292     query_str = "INSERT INTO Running VALUES (" + \
293                 stamp + ", " + \
294                 "'" + command + "'" + ", " + \
295                 "'" + (sys.pairs)["host"] + "'" + ", " + \
296                 str(time.time()) + ", " + \
297                 str(schema.MQ_RUNNING) + ", " + \
298                 "'" + attrs + "'" + ", " + \
299                 "NULL" + ", " + \
300                 str(time.time()) + ", " + \
301                 "'" + (sys.pairs)["sched"] + "'" + ", " + \
302                 "'" + (sys.pairs)["sched_id"] + "'" + ", " + \
303                 "'" + (sys.pairs)["submit_host"] + "'" + ", " + \
304                 str(time.time()) + \
305                 ")"
306     myquery(query_str)
```

Figure D.1: Listing of `server.py`: The module used by both `mqnextjob` and `mqdonejob`.

```
 1  #!/usr/bin/python
 2
 3  # Written by Mark Goldenberg, April 2002
 4
 5  import sys
 6  import pg
 7  import os
 8  import string
 9  import time
10  from Bin import common
11  from Bin import schema
12  from Bin.common import myquery
13  from Bin.common import FAILURE_EXIT_STATUS
14  import server
15
16  def schedule():
17      #debugging
18      query_str = "SELECT * FROM Cache where j_id NOT IN (select j_id from running)"
19      query_result = myquery(query_str)
20
21      query_str = "SELECT * FROM Cache WHERE " + \
22                  "j_id NOT IN (select j_id from running) AND " + \
23                  "hostname = " + "'" + (sys.pairs)["host"] + "'"
24      query_result = myquery(query_str)
25
26      try:
27          cacheTuple = query_result.dictresult()[0]
28      except:
29          query_str = "SELECT * FROM Cache WHERE " + \
30                      "j_id NOT IN (select j_id from running) AND " + \
31                      "hostname = ''"
32          query_result = myquery(query_str)
33          try:
34              cacheTuple = query_result.dictresult()[0]
35          except:
36              server.noJobs()
37
38      jobId = cacheTuple['j_id']
39
40      return jobId
41
42  common.getConnection(schema.DBNAME, schema.HOST,
43                       schema.LOCK_TABLE)
44
45  #FAULT TOLERANCE TEST 1
46  #if (int(time.time()) % 100 < 50):
47  #    sys.exit(FAILURE_EXIT_STATUS)
48  #-------------------------------
49
50  j_id = schema.checkRunning()
51  if j_id != 0:
52      print j_id
53      common.closeConnection()
54      sys.exit(0)
55
56  query_str = "SELECT * FROM Cache where j_id NOT IN (select j_id from running)"
57  query_result = myquery(query_str)
58
59  if query_result.ntuples() == 0:
60      server.addToCache(server.restoreCache())
61
62  query_str = "SELECT * FROM Cache where j_id NOT IN (select j_id from running)"
63  query_result = myquery(query_str)
64
65  if query_result.ntuples() == 0:
66      server.noJobs()
67
68  while 1:
```

```
69      query_str = "SELECT * FROM Cache, Jobs WHERE Cache.j_id = Jobs.j_id" + \
70                  " AND " + "Jobs.comm_line = 'echo'"
71      query_result = myquery(query_str)
72
73      if (query_result.ntuples() == 0):
74          break
75      replaceSet = query_result.dictresult()
76      for id in replaceSet:
77          id = id['j_id']
78          #print id, "is done!"
79          server.done(id, "")
80
81  stamp = schedule()
82
83  print stamp
84  server.setStartTime(stamp)
85  server.next(str(stamp))
86
87  common.closeConnection()
88
89  log_str = "echo " + str(stamp) + " " + str(time.time()) + " " + \
90            ">> ~/starts.log"
91  os.system(log_str)
92
93  log_str = "echo " + str(stamp) + " " + (sys.pairs)["host"] + " " + \
94            ">> ~/hosts.log"
95  os.system(log_str)
96  sys.exit(0)
```

Figure D.2: Listing of `mqnextjob.py`.

```
 1  #!/usr/bin/python
 2
 3  # Written by Mark Goldenberg, April 2002
 4
 5  import sys
 6  import pg
 7  import os
 8  import string
 9  import time
10  from Bin import common
11  from Bin import schema
12  from Bin.common import myquery
13  import server
14
15  common.getConnection(schema.DBNAME, schema.HOST,
16                       schema.LOCK_TABLE)
17
18  stamp = sys.pairs["id"]
19  jobId = int(stamp)
20
21  query_str = "SELECT * FROM Running WHERE j_id = " + \
22              stamp
23  query_result = myquery(query_str)
24
25  if query_result.ntuples() == 0:
26      print "done_job.py: Probably wrong stamp; exiting..."
27      sys.exit(0)
28
29  # The only thing that we need from the last query is the hostname
30  host = query_result.dictresult()[0]["hostname"]
31
32  command = server.done(jobId, host)
33  server.setEndTime(jobId)
```

```
34  server.setHost(jobId, host)
35
36  if command != "echo":
37      log_str = "echo " + stamp + " " + str(time.time()) + " " + ">> ~/ends.log"
38      os.system(log_str)
39
40  common.closeConnection()
41
42  sys.exit(0)
```

Figure D.3: Listing of `mqdonejob.py`.

Appendix E

The Experiment with Data Movement

In this appendix, we show that data movement can be performed within the framework of TrellisDAG and placeholder scheduling.

We start by describing how data movement can be organized and encoded in the placeholder. Then we run experiments with data movement followed by experiments in which data movement is combined with placement affinity for verification jobs.

E.1 Organization of data movement

In this section, we describe how the problem of data transfer was addressed.

The local directory for the checkers computation contains two main directories: `Bin/` and `DataBases/`. All the executables can be found in `Bin/`, while the compressed databases (see Chapter 3) are accumulated in `DataBases/`. We put the precomputed 2, 3, 4, 5, and 6-piece databases into `DataBases/` and predistribute these two directories and all other directories and files that are necessary for the computation to the `/usr/scratch/checkers` directory on each of the nodes in the cluster.[1]

When a new part of the checkers databases is computed, some new files are formed in the local directory. For example, assume that the part 2122.66 is com-

[1] Our system does not make an assumption that the locations of the application on the computational nodes are same; we did it in order to easily automate the experiments.

puted. Then we get 3 new files:

1. `7/2122/BASE2122.66 -- the uncompressed databases.`
2. `7/2122/BASE2122.66.v1 -- the compressed databases.`
3. `7/2122/BASE2122.66.v1.idx -- the index file for the compressed databases.`

We need to make sure that the built databases with all prerequisite slices are locally available to every computation. We have two choices:

1. Maintain a central version of the built databases. Whenever a computation is ready to start, it has to obtain a global lock, move the built databases to the local directory, and release the lock. When the computation is done, it has to obtain a global lock, copy again the central version of the built databases, add the computed databases, copy the updated built databases to the central location, and finally release the lock. The need for global locking substantially reduces the benefit of concurrency.
2. Have each execution node maintain its own copy of the built databases. We will show that this approach can be implemented without using the tools for locking that our system provides.

We choose the second approach. Each execution node has its own copy of the built databases. Whenever a computation is about to start on an execution node, the parts of the databases that have been computed between the previous computation and the current one are copied from the central location and integrated into the local copy of the built databases. When a computation is finished, the computed uncompressed databases are copied to the central location. In order to completely avoid locking, the following scheme is used:

1. At the central location, there is a directory for each execution node: `jasper-00/`, `jasper-01/`, etc. As well, there is a directory `jasper-reserve/` that is used to accumulate all computed parts of the checkers databases.

Experiment #	**1**	2	3
Makespan, sec.	**33999.46**	33998.90	34055.44

Table E.1: The makespans for the large-size experiment with data movement. The median run is shown in bold.

2. Whenever a computation is finished, the computed parts of the databases are built into the local built databases and the two files corresponding to the compressed form of the newly computed databases are copied to the `jasper-reserve/` directory in the central location. Links to the newly created copies are created in all the `jasper-xx` directories except for the one corresponding to the node that completed the computation. Then the local copy of the computed databases is removed. We use the `md5sum` utility in order to make sure that the data has been copied correctly (see the `getData` function of the placeholder, Appendix A).

3. Suppose that a computation is about to start on `jasper-`i. Then all the files are copied from the `jasper-`i`/` directory in the central location to the local directory. Upon success (i.e. when md5sum reports no errors) the copied databases are integrated into the local built databases and removed from both the local and the central location.

The described mechanism is implemented in the functions `getData` and `sendData` of the placeholder (see Appendix A).

E.2 The experiment with data movement and no placement affinity

Table E.1 shows the makespans obtained in the runs. All of the presented charts correspond to the median run #1. The largest deviation from the makespan of the median run is $\frac{34055.44-33999.46}{33999.46} \approx 0.0016 = 0.16\%$.

The flow of computation is summarized with the resource utilization chart in Figure E.1 and the degree of concurrency chart in Figure E.2.

Let us compare this experiment with the experiment without data movement

and see how the total computation time was affected by the introduction of data movement into the picture. Recall that the makespan of the median run without data movement was 32857.36 seconds (see Table 6.9). Then the median run of the current experiment with data movement is only $\frac{33999.46-32857.36}{32857.36} \approx 0.0348 = 3.48\%$ slower than the corresponding run without data movement.

Finally, the charts in Figures E.1 and E.2 look very similar to the corresponding charts in Figures 6.37 and 6.39.

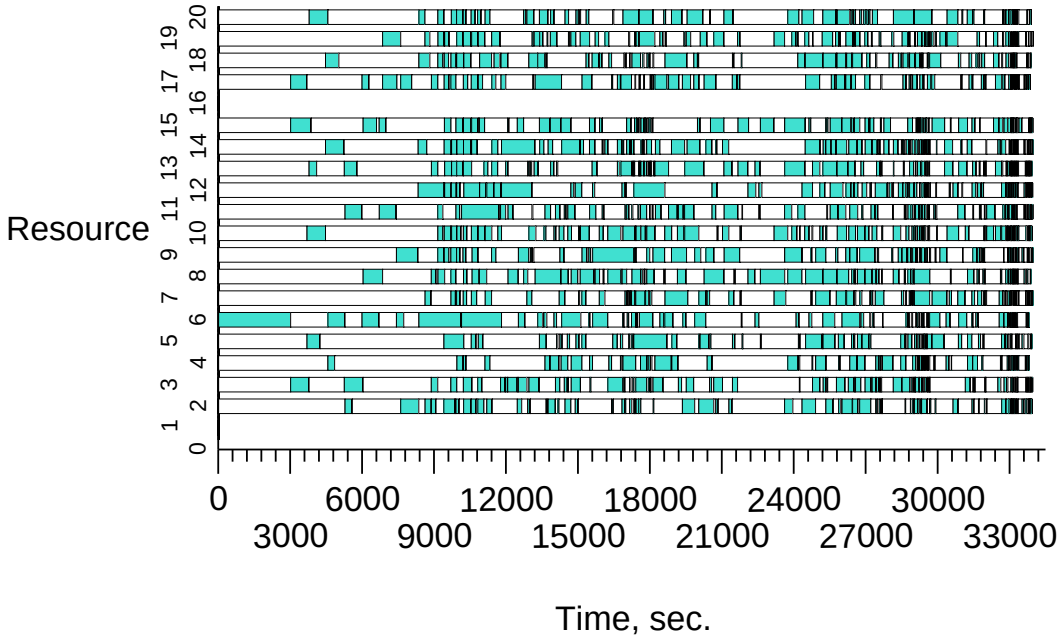

Figure E.1: Resource utilization chart for the large-size experiment with data movement.

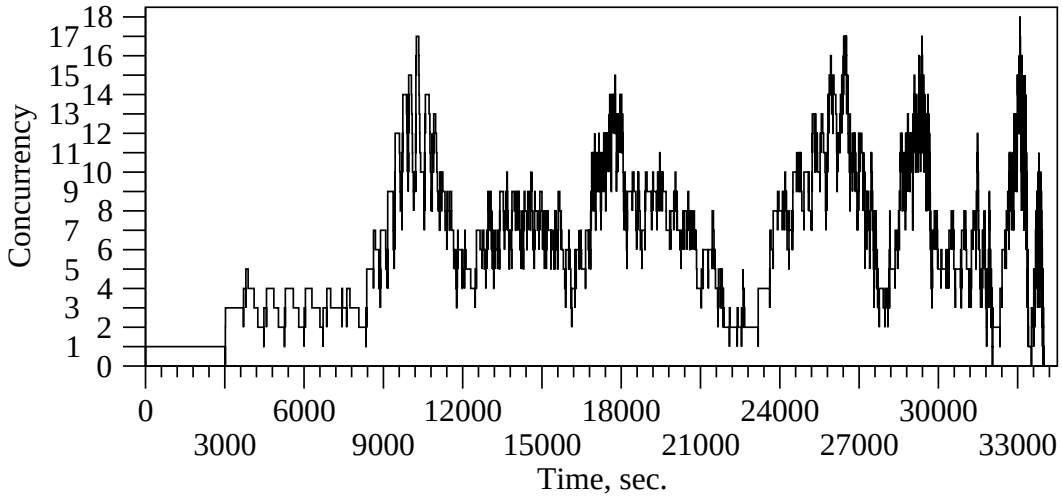

Figure E.2: Degree of concurrency chart for the large-size experiment with data movement.

Experiment #	1	2	**3**
Makespan, sec.	33899.57	34057.43	**34053.61**

Table E.2: The makespans for the large-size experiment with data movement and placement affinity. The median run is shown in bold.

E.3 The experiment with data movement and placement affinity

The discussion of this section is laid out as follows. We start by reminding the reader how the introduction of placement affinity can reduce the makespan. Then, we present the experimental results and note that the makespan has actually increased in comparison to the makespan achieved in the experiment without placement affinity. Finally, we explain why this result is possible and give some numbers that justify our speculations.

Placement affinity can result in improved makespan because it allows reuse of data produced by one job by the next job in the same group. For example, a verification job only needs the data produced by the preceding computation job and, if the verification job is executed on the same machine where the corresponding computation job just completed, then there is no need to obtain any data from the central location.

Table E.2 shows the makespans obtained in the runs. All of the presented charts correspond to the median run #3. The largest deviation from the makespan of the median run is $\frac{34053.61-33899.57}{34053.61} \approx 0.0045 = 0.45\%$.

The flow of computation is summarized with the resource utilization chart in Figure E.3 and the degree of concurrency chart in Figure E.4.

Let us first compare the makespans achieved in this experiment with the makespans achieved in the experiments without placement affinity (i.e. compare Tables E.2 and E.1). The computation with placement affinity is marginally slower than the computation without placement affinity. For the median runs, the computation with placement affinity is $\frac{34053.61-33999.46}{33999.46} \approx 0.0016 = 0.16\%$ slower than the computation without placement affinity.

Before trying to explain this result, we note that the difference is really marginal

and can be attributed to experimental error.

1. The databases produced by each computation job end up being copied to 17 out of 18 execution hosts. Hence, the affinity only saves $1/18$ of the copying time. Let us estimate this time. We use the median runs for the experiment without data movement and the experiment with data movement and no placement affinity to estimate the overhead related to data movement at $33999.46 - 32857.36 = 1142.1$ seconds (see Tables 6.9 and E.1) and thus our savings cannot be higher than $1142.1/18 = 63.45$ seconds. Furthermore, the introduction of placement affinity does not save any time on integrating the newly computed databases into the local built databases, which further reduces the savings.

2. By the very definition of placement affinity, a job with a set affinity attribute can only be given out for execution to a placeholder that runs on the same machine on which the previous job of the job's group was run. Suppose that j_1 and j_2 are consecutive jobs in the same group and the affinity attribute of j_2 is set. When j_1 is completed (i.e. `mqdonejob` has been invoked), there may be a placeholder that is already running on some other machine and is about to invoke `mqnextjob`. Thus, without placement affinity, the gap between the completion time of j_1 and the start time j_2 could have been close to 0. With placement affinity, however, the placeholder that hosted the execution of j_1 has to complete and some other placeholder has to be scheduled on the same machine before j_2 can start. Given the large number of jobs in the experiment, this overhead can accumulate.

Still using placement affinity can be advantageous. The first of our speculations assumes our model of data movement, where a computed part of checkers databases is copied to each of the execution nodes disregarding of whether this part will ever be used on that particular node or not. A different data movement model can be used for other applications, thereby making the savings due to placement affinity much greater. Our model of data movement is justified for the checkers application because:

1. There is a large chance for each node to partake in computing the last `0043` slice of the 4 versus 3 pieces databases, whence it is likely that each node is going to require almost all parts of the databases,

2. The model is simple.

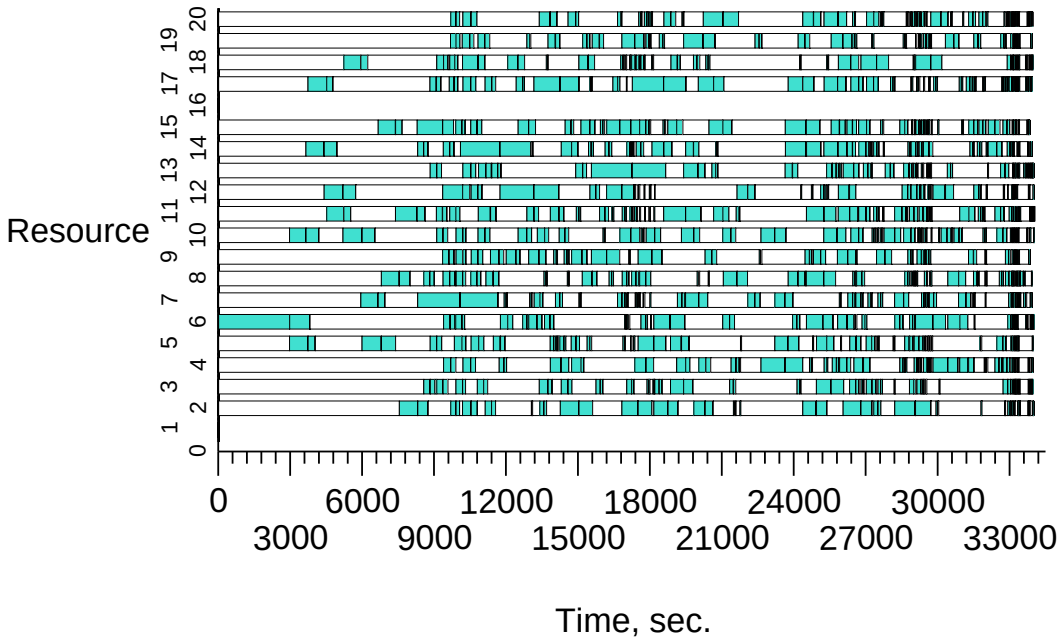

Figure E.3: Resource utilization chart for the large-size experiment with data movement and placement affinity.

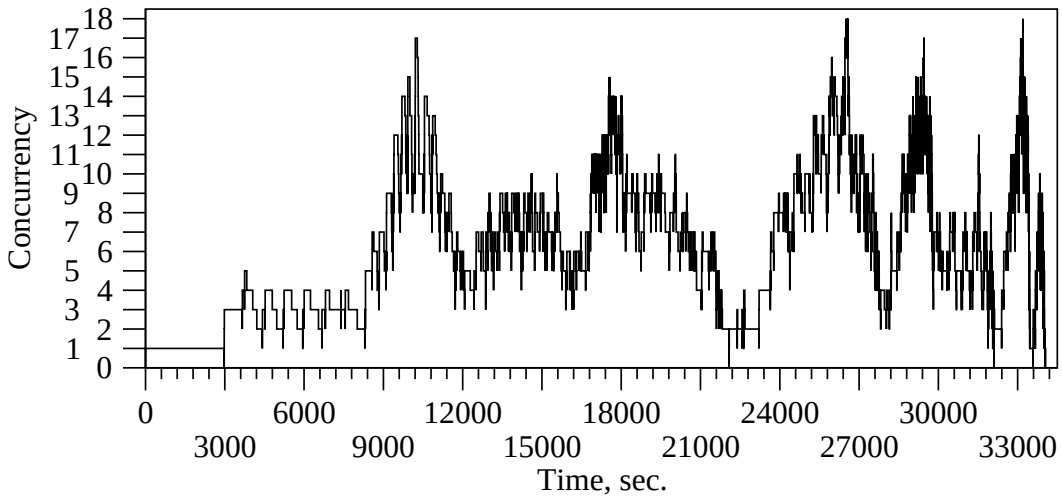

Figure E.4: Degree of concurrency chart for the large-size experiment with data movement and placement affinity.

Appendix F

The Fault Tolerance Daemon

```
 1  #!/usr/bin/python
 2
 3  from Bin import common
 4  from Bin import schema
 5  import os
 6  import sys
 7  import time
 8  import string
 9  from Bin.common import myquery
10
11  def mail(message):
12      command = "ssh -i ~/.ssh/temp brule \"echo " + message + \
13                " | mail -s 'DAG Computation' 'whoami'\""
14      os.system(command);
15
16  MAX_NO_HEAR_TIME = 60 # Have to hear from a job at least this often
17  MAX_TOTAL_TIME = 7200 # No job should run longer than this
18  STOP_TIME = 100 # stop if there is nothing running for this many seconds
19  last = time.time()
20
21  mail("Daemon started")
22  common.getConnection(schema.DBNAME, schema.HOST)
23  while [1]:
24      time.sleep(10)
25
26      query_str = "SELECT j_id FROM Running"
27      result = myquery(query_str)
28      if result.ntuples() == 0:
29          if time.time() - last > STOP_TIME:
30              break
31      else:
32          last = time.time()
33
34      query_str = "SELECT j_id FROM Running WHERE " + \
35                  "float8larger(user_started, last_signal) < " + \
36                  str(time.time() - MAX_NO_HEAR_TIME)
37      result1 = myquery(query_str)
38
39      query_str = "SELECT j_id FROM Running WHERE " + \
40                  "started < " + \
41                  str(time.time() - MAX_TOTAL_TIME)
42      result2 = myquery(query_str)
43
44      ids = []
45
46      for entry in result1.dictresult():
47          ids = ids + entry.values()
48          message = "Did not hear from job " + str(entry.values()) + " for " + \
```

```
49                  str(MAX_NO_HEAR_TIME) + " seconds."
50          mail(message)
51
52      for entry in result2.dictresult():
53          ids = ids + entry.values()
54          message = "Job " + str(entry.values()) + " has been running for " + \
55                    str(MAX_TOTAL_TIME) + " seconds."
56          mail(message)
57
58      if ids != []:
59          os.system("~/build/Bin/mq/suspend.sh") # suspend all placeholders
60          mail("All placeholders are suspended and daemon is exiting due to errors...")
61          common.closeConnection()
62          sys.exit(0)
63
64  common.closeConnection()
65
66  mail("Daemon is done")
```

Figure F.1: The fault tolerance daemon