TrellisDAG: A System for Structured DAG Scheduling

Mark Goldenberg, Paul Lu, and Jonathan Schaeffer

Department of Computing Science
University of Alberta
Edmonton, Alberta, T6G 2E8
Canada
`{goldenbe,paullu,jonathan}@cs.ualberta.ca`
`http://www.cs.ualberta.ca/~paullu/Trellis/`

Abstract. High-performance computing often involves sets of jobs or workloads that must be scheduled. If there are dependencies in the ordering of the jobs (e.g., pipelines or directed acyclic graphs) the user often has to carefully, manually submit the jobs in the right order and/or delay submitting dependent jobs until other jobs have finished. If the user can submit the *entire* workload with dependencies, then the scheduler has more information about future jobs in the workflow.
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. TrellisDAG also has a modular architecture for implementing different scheduling *policies*, which will be the topic of future work. Currently, TrellisDAG supports:

1. A spectrum of mechanisms for users to specify both simple and complicated workflows.
2. The ability to load balance across multiple administrative domains.
3. A convenient tool to monitor complicated workflows.

1 Introduction

High-performance computing (HPC) often involves sets of jobs or workloads that must be scheduled. Sometimes, the jobs in the workload are completely independent and the scheduler is free to run any job concurrently with any other job. At other times, the jobs in the workload have application-specific dependencies in their ordering (e.g., pipelines [20]) such that the user has to carefully submit the jobs in the right order (either manually or via a script) or delay submitting dependent jobs until other jobs have finished.

The details of which job is selected to run on which processor is determined by a *scheduling policy* [5]. Often, the scheduler uses the knowledge of the jobs in the submission queue, the jobs currently running, and the history of past jobs to help make its decisions. In particular, any knowledge about future job arrivals can supplement other knowledge to make better policy choices.

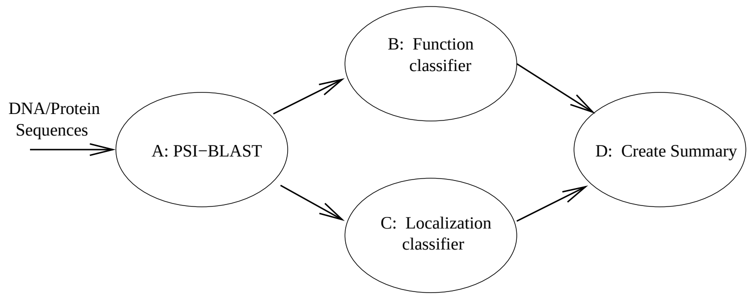

Fig. 1. Example Bioinformatics Workflow

The problem is that the mere presence of a job in the submission queue is usually interpreted by the scheduler to mean that a job can run concurrently with other jobs. Therefore, to avoid improper ordering of jobs, either the scheduler has to have a *mechanism* to specify job dependencies or the user has to delay the submission of some jobs until other jobs have completed. Without such mechanisms, managing the workload can be labour-intensive and deprives the scheduler of the knowledge of future jobs until they are actually submitted, even though the workflow "knows" that the jobs are forthcoming. If the user can submit the *entire* workload with dependencies, then the scheduler has access to more information about future jobs in the workflow.

We have designed, implemented, and evaluated the TrellisDAG system for scheduling workloads with job dependencies [10]. TrellisDAG is designed to support any workload with job dependencies and provides a framework for implementing different scheduling policies. So far, our efforts have focussed on the basic mechanisms to support the scheduling of workflows with directed acyclic graph (DAG) dependencies. So far, our policies have been simple: first-come-first-serve (FCFS) of jobs with satisfied dependencies and simple approaches to data locality when placing jobs on processors [10]. We have not yet emphasized the development of new policies since our focus has been on the underlying infrastructure and framework.

The main contributions of this work are:

1. The TrellisDAG system and some of its novel *mechanisms* for *expressing job dependencies*, especially DAG description scripts (Section 3.2).
2. The description of *an application with non-trivial workflow dependencies*, namely building checkers endgame databases via retrograde analysis (Section 2).
3. A simple, *empirical evaluation* of the correctness and performance of the TrellisDAG system (Section 4).

1.1 Motivation

Our empirical evaluation in Section 4 uses examples from computing checkers endgame databases (Section 2). However, TrellisDAG is designed to be general-purpose and to support a variety of applications. For now, let us consider a bioinformatics application with a simple workflow dependency with four jobs: A, B, C, and D (Figure 1). This example is based on a bioinformatics research project in our department called Proteome Analyst (PA) [19].

In Figure 1, the input to the workflow is a large set of DNA or protein sequences, usually represented as strings over an alphabet. In high-throughput proteome analysis, the input to Job A can be tens of thousands of sequences. A common, first-stage analysis is to use PSI-BLAST [1] to find similar sequences, called homologs, in a database of known proteins. Then, PA uses the information from the homologs to predict different properties of the new, unknown proteins. For example, Job B uses a machine-learned classifier to map the PSI-BLAST output to a prediction of the general function of the protein (e.g., the protein is used for amino acid biosynthesis). Job C uses the same PSI-BLAST output from A and a different classifier to predict the subcellular localization of the protein (e.g., the protein is found in the Golgi complex). Both Jobs B and C need the output of Job A, but both B and C can work concurrently. For simplicity, we will assume that all of the sequences must be processed by PSI-BLAST before any of the output is available to B and C. Job D gathers and presents the output of B and C.

Some of the challenges are:

1. Identifying that Jobs B and C can be run concurrently, but A and B (and A and C) cannot be concurrent (i.e., knowledge about dependencies within the workflow).
2. Recognizing that there are, for example, three processors (not shown) at the moment that are ready to execute the jobs (i.e., find the processor resources).
3. Mapping the jobs to the processors, which is the role of the scheduler.

Without the knowledge about dependencies in the scheduler, the user may have to submit Job A, wait until A is completed, and then submit B and C so that the ordering is maintained. Otherwise, if Jobs A, B, C, and D are all in the submission queue, the scheduler may try to run them concurrently if, say, four processors are available. But, having the user manually wait for A to finish before submitting the rest of the workflow could mean delays and it does mean that the scheduler cannot see that B, C, and D will eventually be executed, depriving the policy of that knowledge of future jobs.

1.2 Related work and context

The concept of grid computing is pervasive these days [7, 6]. TrellisDAG is part of the Trellis Project in overlay metacomputing [15, 14]. In particular, TrellisDAG is layered on top of the placeholder scheduling technique [14]. The goals

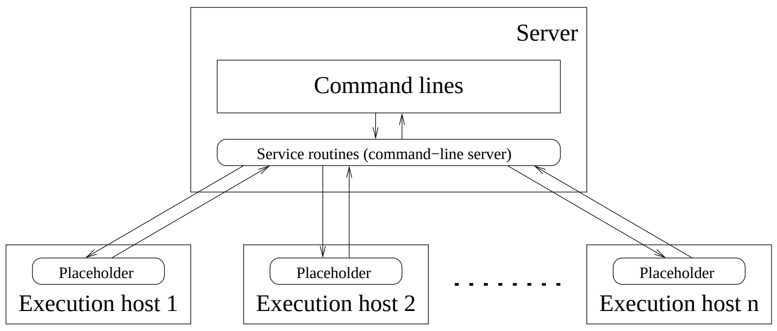

Fig. 2. Placeholder scheduling.

of the Trellis Project are more modest and simpler than that of grid computing. Trellis is focussed on supporting scientific applications on HPC systems; supporting business applications and Web services are not explicit goals of the Trellis Project. Therefore, we prefer to use older terminology—metacomputing—to reflect the more limited scope of our work.

In computing science, the dream of metacomputing has been around for decades. In various forms, and with important distinctions, it has also been known as distributed computing, batch scheduling, cycle stealing, high-throughput computing, peer-to-peer systems, and (most recently) grid computing. Some well-known, contemporary examples in this area include SETI@home [18], Project RC5/distributed.net [16], Condor [13, 8, 3], and the projects associated with Globus/Open Grid Service Architecture (OGSA) [9]. Of course, there are many, many other related projects around the world.

The Trellis philosophy has been to write the minimal amount of new software and to require the minimum of superuser support. Simplicity and software reuse have been important design principles; Trellis uses mostly widely-deployed, existing software systems. Currently, Trellis does not use any of the new software that might be considered part of grid computing, but the design of Trellis supports the incorporation of and/or co-existence with grid technology in the future.

At a high level, placeholder scheduling is illustrated in Figure 2. A *placeholder* is a mechanism for global scheduling in which each placeholder represents a potential unit of work. The current implementation of placeholder scheduling uses normal batch scheduler job scripts to implement a placeholder. Placeholders are submitted to the local batch scheduler with a normal, non-privileged user identity. Thus, local scheduling policies and job accounting are maintained.

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 storage format is user-defined; it can be a plain file, a database, or any other format). There is also a set of separate programs called *services* that form a layer called the *command-line server*. Adding a new storage format or implementing a new scheduling policy corresponds to implementing a new service program in this modular architecture.

There are a 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 placeholder(s) running. Placeholders can handle either sequential or parallel jobs. It can be implemented as a shell script, a binary executable, or a script for a local batch scheduler. 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 (if necessary).

Therefore, when the job (a.k.a. placeholder) begins executing, it contacts a central server and requests the job's actual run-time parameters (i.e., late binding). For placeholders, the communication across administrative domains is handled using Secure Shell (SSH). In this way, a job's parameters are *pulled* by the placeholder rather than *pushed* by a central authority. In contrast, normal batch scheduler jobs hard-code all the parameters at the time of local job submission (i.e., early binding).

Placeholder scheduling is similar to Condor's *gliding in* and *flocking* techniques [8]. Condor is, by far, the more mature and robust system. However, by design, placeholders are not as tightly-coupled with the server as Condor daemons are with the central Condor servers (e.g., no I/O redirection to the server). Also, placeholders use the widely-deployed SSH infrastructure for secure and authenticated communication across administrative domains. The advantage of the more loosely-coupled and SSH-based approach is that overlay metacomputers (which are similar to "personal Condor pools") can be quickly deployed, without superuser permissions, while maintaining functionality and security.

Recently, we used placeholder scheduling to run a large computational chemistry application across 18 different administrative domains, on 20 different systems across Canada, with 1,376 processors [15, 2]. This experiment, dubbed the Canadian Internetworked Scientific Supercomputer (CISS), was most notable for the 18 different administrative domains. No system administrator had to install any new infrastructure software (other than SSH, which was almost-universally available already). All that we asked for was a normal, user-level account. In terms of placeholder scheduling, most CISS sites can be integrated within minutes. We believe that the low infrastructure requirements to participate in CISS was key in convincing such a diverse group of centres to join in.

1.3 Unique features of TrellisDAG

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

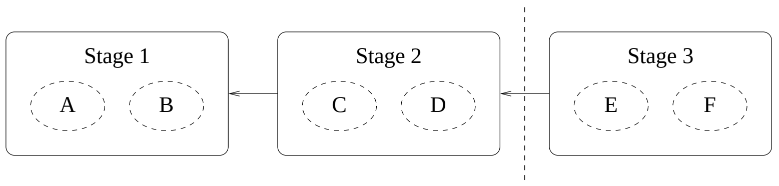

Fig. 3. A 3-stage computation. It is convenient to inquire the status of each stage. If the second stage resulted in errors, we may want to disable the third stage and rerun starting from the second stage.

for the application domain. As shown in Figure 3, suppose 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. The following are examples of the natural tasks of monitoring and administering such a computation:

1. Query the status of the first stage of the computation, e.g. is it completed?
2. Make the system execute only the first two stages of the computation, in effect disabling the third stage.
3. Redo 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 (e.g., Stage 1) and specify collective dependencies between the defined groups (e.g., between Stage 1 and Stage 2).

Finally, TrellisDAG provides mechanisms such that:

1. The user can submit all of the jobs and dependencies at once.
2. TrellisDAG provides a single point for describing the scheduling policies.
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.
4. TrellisDAG records the history information associated with the workflow. For our future work, the scheduler may use machine 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.

Notably, Condor has a tool called the *Directed Acyclic Graph Manager* (*DAGMan*) [4]. One can represent a hierarchical system of jobs and dependencies using DAGMan scripts. DAGMan and TrellisDAG share similar design goals. However, DAGMan scripts are approximately as expressive as TrellisDAG's `Makefile`-based mechanisms (Section 3.2). It is not clear how well DAGMan's scripts will scale for complicated workflows, as described in the next section.

Also, the widely-used Portable Batch System (PBS) has a simple mechanism to specify job dependency, but jobs are named according to submit-time job numbers, which are awkward to script, re-use, and do not scale to large workflows.

2 The Endgame Databases Application

We introduce an important 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. But, the design of TrellisDAG does not make any checkers-specific assumptions.

A team of researchers in the Department of Computing Science at the University of Alberta aims to solve the game of checkers [11, 12, 17]. For this paper, the detailed rules of checkers are largely irrelevant. In terms of workflow, the key application-specific "Properties" are:

1. The game starts with 24 pieces (or checkers) on the board. There are 12 black pieces and 12 white pieces.
2. Pieces are captured and removed from the board during the game. Once captured, a piece cannot return to the board.
3. Pieces start as checkers but can be promoted to be kings. Once a checker is promoted to a king, it can never become a checker again.

Solving the game of checkers means that, for any given position, the following question has to be answered: *Can the side to move first force a win or is it a draw?* Using retrograde analysis, a database of endgame positions is constructed [12, 17]. Each entry in the database corresponds to a unique board position and contains one of three possible values: WIN, $LOSS$ or $DRAW$. Such a value represents perfect information about a position and is called the *theoretical value* for that position.

The computation starts with the trivial case of one piece. We know that whoever has that last piece is the winner. If there are two pieces, any position in which one piece is immediately captured "plays into" the case where there is only one piece, for which we already know the theoretical value. 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 will 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 the 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).

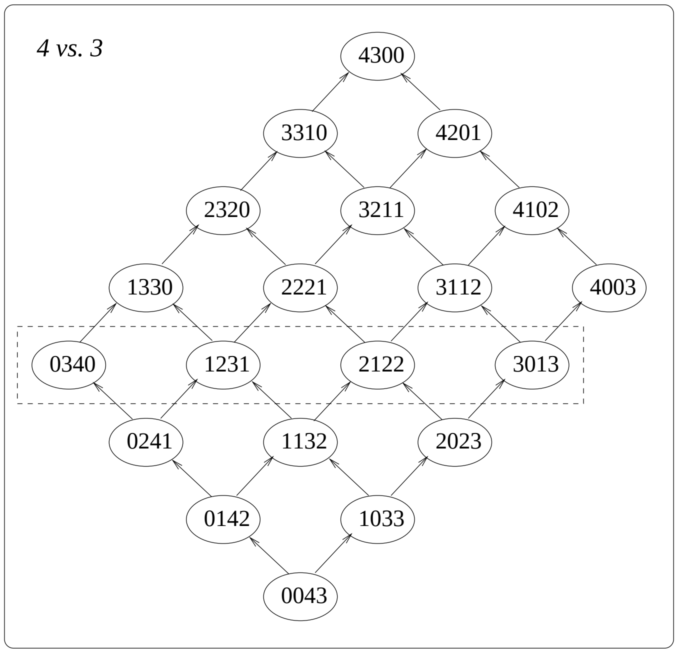

Fig. 4. Example workflow dependencies in checkers endgame databases, part of the 7 piece database.

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 $O(10^{20})$ possible positions in the game. That is why the retrograde analysis is combined with the forward search, in which the 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.

In terms of workflow, strictly speaking, the positions with fewer pieces on the board must be computed/solved before the positions with more pieces. In a position with 3-pieces, a capture immediately results in a 2-piece position, as per Property 2. In other words, the 2-piece database must be computed before the 3-piece databases, which are computed before the 4-piece databases, and so on until, say, the 7-piece databases.

We can subdivide the databases further. Suppose that black has 4 pieces and white has 3 pieces, which is part of the 7-piece database (Figure 4). We note that a position with 0 black kings, 3 white kings, 4 black checkers, and no white checkers (denoted `0340`) can never play into a position with 1 black king, 2 white kings, 3 black checkers, and 1 white checker (denoted `1231`) or vice versa. As per Property 3, `0340` cannot play into `1231` because the third **white** king in `0340` can never become a checker again. `1231` cannot play into `0340` because the lone **black** king in `1231` can never become a checker again. If `0340` and `1231` are separate computations or jobs, they can be computed concurrently as long as the database(s) that they *do* play into (e.g., `1330`) are finished.

In general, let us denote any position by four numbers standing for the number of kings and checkers of each color on the board, as illustrated above. A group of all positions that have 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:

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

$$\text{or}$$

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

Figure 4 shows the workflow dependencies for the case of 4 pieces versus 3 pieces. Each "row" in the diagram (e.g., the dashed-line box) represents a set of jobs that can be computed in parallel.

In turn, each of the slices in Figure 4 can be further subdivided (but not shown in the figure) by considering the *rank* of the leading checker. The rank is the row of a checker as it advances towards becoming a king, as per Property 3. Only the rank of the leading or most advanced checker is currently considered. Ranks are numbers from 0 to 6, since a checker would be promoted to become a king on rank 7. For example, `1231.53` represents a position with 1 black king, 2 white kings, 3 black checkers, and 1 white checker, where the leading black checker is on rank 5 (i.e., 2 rows from being promoted to a king) and the leading white checker is on rank 3 (i.e., 4 rows from being promoted to a king). Consequently, slices where only one side has a checker (and the other side only has kings) have seven possible subdivisions based on the rank of the leading checker. Slices where both sides have checkers have 49 possible subdivisions. Of course, slices with only kings cannot be subdivided according to the leading checker strategy.

To summarize, subdividing a database into thousands of jobs according to the number of pieces, then the number of pieces of each colour, then the combination of types of pieces, and by the rank of the leading checker has two main benefits: an increase in inter-job concurrency and a reduction in the memory and computational requirements of each job. We emphasize that the constraints of 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.

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:

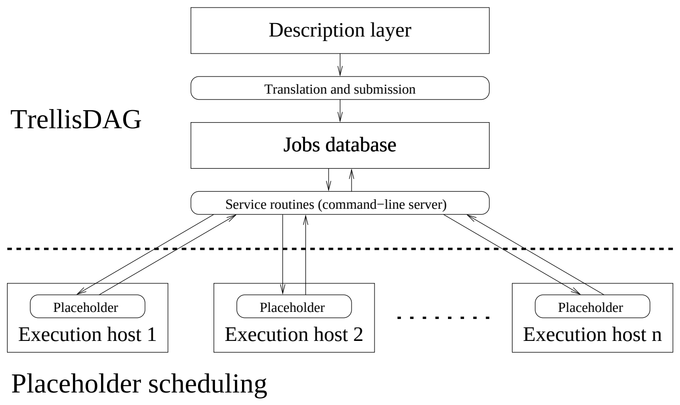

Fig. 5. An architectural view of TrellisDAG.

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

3 Overview of TrellisDAG

An architectural view of TrellisDAG is presented in Figure 5. To run the computation, the user has to submit the jobs and the dependencies to the *jobs database* using one of the several methods described in Section 3.2. 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 services of the command-line server are described in Section 3.3. Finally, the monitoring and administrative utilities are described in Section 3.4.

3.1 The group model

In our system, each job is a 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 either 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 a subgroup is one level lower than its immediate supergroup.

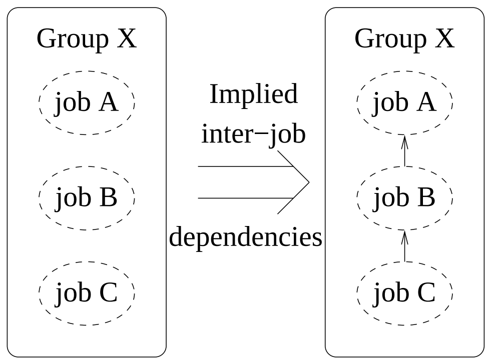

Fig. 6. Dashed ovals denote jobs. The dependencies between the jobs of group X are implicit and determined by the order of their submission.

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 6.
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 7.
3. A supergroup may have jobs of its own. Such jobs are executed after all subgroups of the supergroup are completed. This is illustrated in Figure 8.
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 9.
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 10.

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. The workflow for our example is shown in Figure 11.

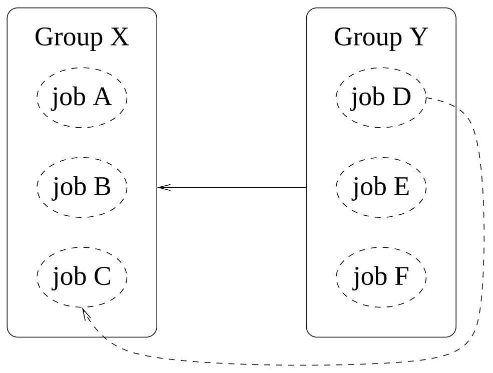

Fig. 7. The workflow dependency of group Y on group X implies the dependency of the first job of Y on the last job of X (this dependency is shown by the dashed arc).

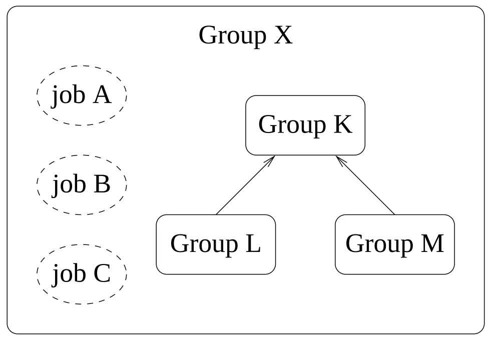

Fig. 8. Group X has subgroups K, L, M and jobs A, B, C. These jobs will be executed after all of the jobs of K, L, M and their subgroups are completed.

We can think of defining supergroups for the workflow in Figure 11 as shown by the dashed lines. Then, we can define dependencies between the supergroups and obtain a simpler looking workflow.

3.2 Submitting the workflow

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.

Flat submission script. The first way of submitting a workflow is by using the `mqsub` utility. This utility is similar to `qsub` of many batch schedulers. However, there is an extra command-line argument (i.e., `-deps`) to `mqsub` that lets the

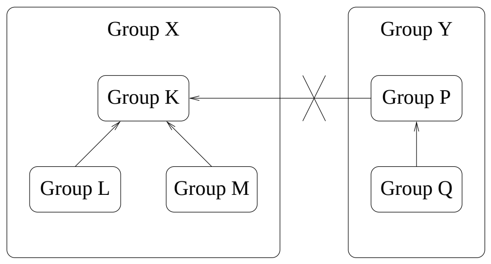

Fig. 9. Groups K, L, M have same supergroup: X; therefore, we can specify dependencies between these groups. Similarly, group Y is a common supergroup for groups P and Q. In contrast, groups K and P do not have a common supergroup (at least not immediate supergroup) and cannot have an explicit dependency between them.

user specify workflow dependencies. Note that the names of groups are user-selected and are not scheduler job numbers; the scripts can be easily re-used. In our example, we define a group for each slice to be computed. The (full-size version of the) script in Figure 12 submits the workflow in Figure 11.

Note that 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 order, and
2. The jobs within a group have to be submitted in the correct order.

Using a Makefile. A higher-level description of a workflow is possible via a Makefile. The user simply writes a Makefile that can be interpreted by the standard UNIX `make` utility. Each rule in this Makefile computes a part of the checkers databases and specifies the dependencies of that part on other parts. TrellisDAG includes a utility, called `mqtranslate`, that translates such a Makefile in another Makefile, in which every command line is substituted for a call to `mqsub`. We present part of a translated Makefile for our example in Figure 13.

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 this framework (which we call the *DAG description script*), the additional functionality of supergroups becomes available.

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

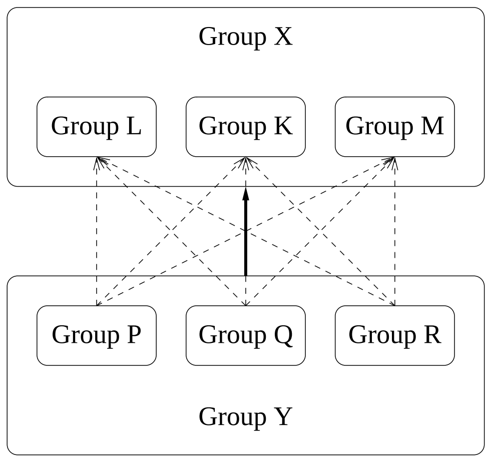

Fig. 10. Group Y depends on group X and this implies pairwise dependencies between their subgroups (these dependencies are denoted by dashed arrows).

The sample DAG description script in Figure 14 describes the workflow in Figure 11 with supergroups. Line 19 states that there are two levels of groups. A group at level `VS` is identified by two integers, while a group at level `Slice` is identified by four integers. The function `generateGroup` (lines 21-41) returns the list of groups with a given supergroup at a given level. The generated groups correspond to the nodes (in case of the level `Slice`) and the dashed boxes (in case of the level `VS`) in Figure 11. The function `getDependent` (lines 57-61) returns a list of groups with a given supergroup, on which a given group with the same supergroup depends. The executables for the jobs of a given group are returned by the `getJobsExecutables` (lines 43-46) function. Note that, in our example, computing each slice involves a computation and a verification job; they are represented by executables with suffixes `.comp.sh` and `.ver.sh`, respectively. We will now turn to describing the `getJobsAttributes` function (lines 48-51).

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 the current version of the system, we have several kinds of attributes that serve the following purposes:

1. Increase the degree of concurrency by relaxing workflow dependencies.
2. Regulate the placement of jobs, i.e. mapping jobs to execution hosts.
3. Store the history profile.

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

By separating the computation of the checkers endgame databases with their verification, we can potentially achieve a higher degree of concurrency. To do that, we introduce an attribute that allows the dependent groups to proceed

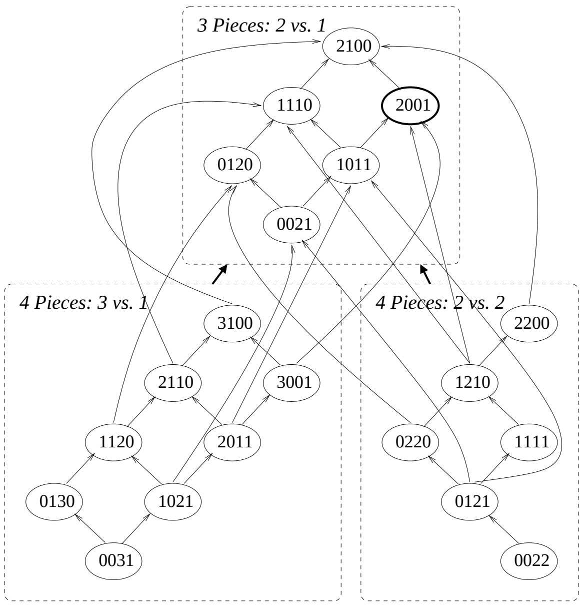

Fig. 11. The workflow for the running example. The nodes are slices. The dashed lines represent the natural way of defining supergroups; if such supergroups were defined, then the only workflow dependencies would be expressed by the thick arrows.

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`.

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.

3.3 Services of the command-line server

Services of the command-line server (see Figure 5) 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 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. For example, the service could be called as follows:

```
     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"
...snip...
```

Fig. 12. Part of the flat submission script for the running example.

```
    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"
...snip...
```

Fig. 13. Part of the Makefile with calls to `mqsub` for the running example.

```
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 placeholder has access to the attributes of a job through the `mqgetjobattribute` service. The service outputs the value associated with the given attribute. When the job is complete, the placeholder has to inform the system about this event. This is done using the `mqdonejob` service.

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 3.4).

3.4 Utilities

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

```
#!/usr/bin/python

import sys
import math
import string

nPieces = int(sys.argv[1])

def getDependentSlice(slice):
    # a checker of one of the colors might have become a king
    return [[slice[0] + 1, slice[1], slice[2] - 1, slice[3]],
            [slice[0], slice[1] + 1, slice[2], slice[3] - 1]]

def getDependentVS(vs):
    # there could be a capture
    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): # loop for number of pieces
            #e.g. for np=4, generate [[3, 1], [2, 2]]
            for b in range((np + 1)/2, np):
                result.append([b, np - b])
    else:
        b = parentGroup[0] # number of black pieces
        w = parentGroup[1] # number of white pieces
        for bk in range(0, b + 1): # loop for black kings
            for wk in range(0, w + 1): # loop for white kings
                # number of checkers is determined
                bc = b - bk
                wc = w - wk
                # savings due to symmetry
                if bk + bc < wk + wc: continue
                if bk + bc == wk + wc and bc < wc: continue
                # a slice is formed
                result.append([bk, wk, bc, wc])
    return result

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)
```

Fig. 14. The DAG description script with supergroups and attributes.

```
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)
```

Fig. 15. The first screen of `mqdump` for the running example.

`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.

`mqdump`**: the 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 14 is presented in Figure 15. Following the prompt, the user types the number from 1 to 11 corresponding to the command that must be executed.

As we see from Figure 15, the user can dive into the supergroups, see the status of the computation of each group, and perform several administrative tasks. For example, the user can disable a group, thereby preventing it from being executed; or, he can assert that a part of the computation was performed without using TrellisDAG by marking one or more groups as done.

3.5 Concluding remarks

The features of TrellisDAG can be summarized as follows (see Figure 5):

1. The workflow and its dependencies can be described in several ways. The user chooses the mechanism or technique 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 by the user to perform monitoring and simple administrative tasks.
5. The history profile of the last computation is stored in the jobs' attributes.

4 Experimental Assessment of TrellisDAG

As a case study, we present an experiment that shows how TrellisDAG can be applied to compute part of the checkers endgame databases. The goal of these simple experiments is to demonstrate the key features of the system, namely:

1. **Simplicity of specifying dependencies.** Simple workflows are submitted using `mqsub`. For other workflows, writing the DAG description script is a fairly straightforward task (Figure 14), considering the complexity of the dependencies.
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. Since TrellisDAG is bound to placeholder scheduling, this overhead includes the overhead of running the placeholders.

We factored out the data movement overheads in these experiments. However, data movement can be supported [10].

4.1 The minimal schedule

We introduce our concept of the *minimal schedule*. We use this concept to tackle the following question: What is considered a good utilization of opportunities for concurrent execution? The 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 the job is run without any start-up overhead. Therefore, the only limits on the degree of concurrency in the minimal schedule are job dependencies (not resources) and there are no scheduler overheads.

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 job, store its estimated (use the time measured during the sequential run) time of completion. 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. 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.

4.2 Experimental setup

All of the experiments were performed using a dedicated cluster of 20 nodes with dual AMD Athlon 1.5GHz CPUs and 512MB of memory. Since the application is memory-intensive, we only used one of the CPUs on each node. Sun Grid Engine (SGE) was the local batch scheduler and the PostgreSQL database was used to maintain the jobs database. After each run, the computed databases are verified against a trusted, previously-computed version of the database. If the ordering of computation in the workflow is incorrect, the database will not verify correctly. Therefore, this verification step also indicates a proper schedule for the workflow.

4.3 Computing all 4 versus 3 pieces checkers endgame databases

We use the grouping capability of TrellisDAG and define 3 levels: `VS` (as in 4 "versus" 3 pieces in the 7-piece database), `Slice` and `Rank` (which are further subdivisions of the problem). Over 3 runs, the median time was 32,987 seconds (i.e., 9.16 hours) (with a maximum of 33,067 seconds and a minimum of 32,891 seconds). The minimal schedule is computed to be 32,444 seconds, which implies that our median time is within 1.7% of the minimal schedule. Since the cluster is dedicated, this result is largely expected. However, the small difference between the minimal schedule and the computation does show that TrellisDAG does make good use of the available concurrency (otherwise, the deviation from the minimal schedule would be much greater). Furthermore, the experiment shows that the cumulative overhead of the placeholders and the TrellisDAG services is small compared to the computation.

The degree of concurrency chart is shown in Figure 16. This chart shows how many jobs were being executed concurrently at any given point in time. We note that the maximal degree of concurrency achieved in the experiment is 18. However, the total time when the degree of concurrency was 18 (or greater than 12 for that matter) is very small and adding several more processors would not significantly affect the makespan.

The degree of concurrency chart in Figure 17 corresponds to the minimal schedule. Note that the maximal degree of concurrency achieved by the minimal schedule is 20 and this degree of concurrency was achieved approximately at

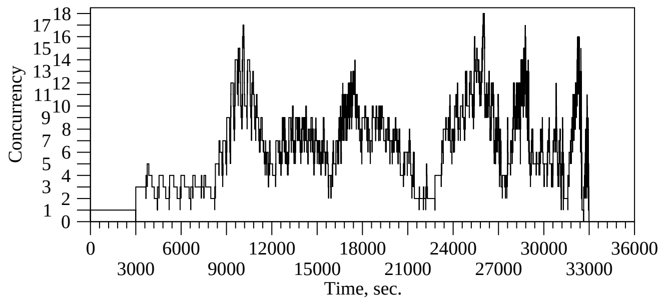

Fig. 16. Degree of concurrency chart for the experiment without data movement.

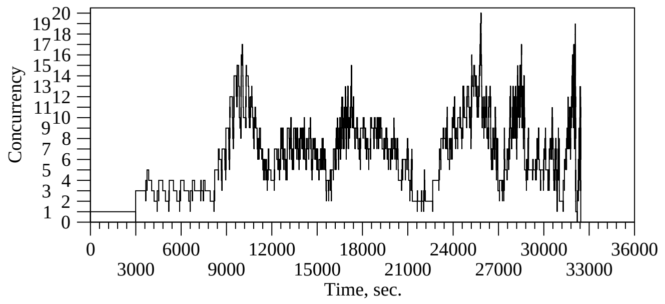

Fig. 17. Degree of concurrency chart for the experiment without data movement. **Minimal schedule**.

the same time in the computation as when the degree of concurrency 18 was achieved in the real run (compare Figure 16 and Figure 17). With this exception, the degree of concurrency chart for the minimal schedule run looks like a slightly condensed version of the chart for the real run.

Summary The given 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. Also, the makespan of the computation was close to that of the model run. Elsewhere, we show that the system can also be used for computations requiring data movement [10].

5 Concluding remarks

We have shown how TrellisDAG can be effectively used to submit and execute a workflow represented by a large DAG of dependencies. So far, our work has concentrated on providing the *mechanisms* and capability for specifying (e.g., DAG description script) and submitting entire workflows. The motivation for our work is two-fold: First, some HPC workloads, from bioinformatics to retrograde analysis, have non-trivial workflows. Second, by providing scheduler mechanisms to describe workloads, we hope to facilitate future work in scheduling *policies* than can benefit from the knowledge of forthcoming jobs in the workflow.

In the meantime, several properties of TrellisDAG have been shown:

1. DAG description scripts for the workflows are relatively easy to write.
2. The makespans achieved by TrellisDAG are close to the makespans achieved by the minimal schedule. We conclude that the system effectively utilizes the opportunities for concurrent execution, and
3. The overhead of using TrellisDAG is not high, which is also justified by comparing the makespan achieved by the computation using TrellisDAG with the makespan of the model computation.

As discussed above, our future work with TrellisDAG will include research on new scheduling policies. We believe that the combination of knowledge of the past (i.e., trace information on service time and data accessed by completed jobs) and knowledge of the future (i.e., workflow made visible to the scheduler) is a powerful combination when dealing with multiprogrammed workloads and complicated job dependencies. For example, there is still a lot to explore in terms of policies that exploit data locality when scheduling jobs in an entire workload.

In terms of deployment, we hope to use TrellisDAG in future instances of CISS [15] and as part of the newly-constituted, multi-institutional WestGrid Project [21] in Western Canada. In terms of applications, TrellisDAG will be a key component in the Proteome Analyst Project [19] and in our on-going attempts to solve the game of checkers.

References

1. S. F. Altschul, T. L. Madden, A. A. Schaffer, J. Zhang, Z. Zhang, W. Miller, and D. J. Lipman. Gapped BLAST and PSI–BLAST: a new generation of protein database search programs. *Nucleic Acids Res.*, 25:3389–3402, 1997.
2. CISS – The Canadian Internetworked Scientific Supercomputer. `http://www.cs.ualberta.ca/~ciss/`.
3. Condor. `http://www.cs.wisc.edu/condor`.
4. DAGMan Metascheduler. `http://www.cs.wisc.edu/condor/dagman/`.
5. 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.

6. I. Foster and C. Kesselman, editors. *The Grid: Blueprint for a Future Computing Infrastructure.* 1999.
7. 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.
8. T. Frey, J.; Tannenbaum, M. Livny, I. Foster, and S. Tuecke. Condor-G: a computation management agent for multi- institutional grids. In *High Performance Distributed Computing, 2001. Proceedings. 10th IEEE International Symposium*, pages 55 – 63, San Francisco, CA, USA, August 2001. IEEE Computer Society Press.
9. Globus Project. `http://www.globus.org/`.
10. M. Goldenberg. TrellisDAG: A System For Structured DAG Scheduling. Master's thesis, Dept. of Computing Science, University of Alberta, Edmonton, Alberta, Canada, 2003.
11. 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.
12. 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.
13. M. J. Litzkow, M. Livny, and M. W. Mutka. Condor : A hunter of idle workstations. In *8th International Conference on Distributed Computing Systems*, pages 104–111, Washington, D.C., USA, June 1988. IEEE Computer Society Press.
14. 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.
15. C. Pinchak, P. Lu, J. Schaeffer, and M. Goldenberg. The Canadian Internetworked Scientific Supercomputer. In *17th Annual International Symposium on High Performance Computing Systems and Applications (HPCS)*, pages 193–199, Sherbrooke, Quebec, Canada, May 11 – 14, 2003.
16. RC5 Project. `http://www.distributed.net/rc5`.
17. J. Schaeffer, Y. Björnsson, N. Burch, R. Lake, P. Lu, and S. Sutphen. Building the Checkers 10-Piece Endgame Databases. In *Advances in Computer Games X*, 2003. in press.
18. SETI@home. `http://setiathome.ssl.berkeley.edu/`.
19. D. Szafron, P. Lu, R. Greiner, D. Wishart, Z. Lu, B. Poulin, R. Eisner, J. Anvik, C. Macdonell, and B. Habibi-Nazhad. Proteome Analyst – Transparent High-Throughput Protein Annotation: Function, Localization, and Custom Predictors. Technical Report TR 03-05, Dept. of Computing Science, University of Alberta, 2003. `http://www.cs.ualberta.ca/~bioinfo/PA/`.
20. D. Thain, J. Bent, A.C. Arpaci-Dusseau, R.H. Arpaci-Dusseau, and M. Livny. The architectural implications of pipeline and batch sharing in scientific workloads. Technical Report 1463, Computer Sciences Department, University of Wisconsin, Madison, 2002.
21. WestGrid. `http://www.westgrid.ca/`.