Scheduling data flow applications using linear programming

Luiz Thomaz do Nascimento Renato A. Ferreira Wagner Meira Jr. Dorgival Guedes

Departament of Computer Science

Universidade Federal de Minas Gerais

Belo Horizonte, MG, Brazil

{lthomaz,renato,meira,dorgival}@dcc.ufmg.br

Abstract

Data intensive applications are becoming more important in many fields of science and engineering. This fact generates a demand for distributed environments for running such applications efficiently. In this context, several initiatives have emerged in Computational Grids, Datacutter being one such environment. In Datacutter, applications are modeled as a set of communicating filters that may run on several nodes of a Computational Grid. To achieve high performance, a number of transparent copies of each of the filters that comprise the application need to be appropriately placed on different nodes of the grid. Such task is carried out by a scheduler which is the focus of this work. We present LPSched, a scheduler for Datacutter applications which uses linear programming to make decisions about the number of copies of each filter as well as the placement of each of the copies across the nodes. LPSched bases its decisions upon the performance behavior of each filter as well as the resources currently available on the Grid.

1 Introduction

With the continuous advances on computational models for many applications in science and engineering, unprecedented amounts of data are being continuously generated. Driven by the evolution in the price of storage equipament more and more of these datasets are being stored for later processing. This is true in many areas, for instance, image processing [1, 2], data mining applications [3] and model simulation applications [4, 5, 6]. The performance requirements for such applications have been pushing the envelope on the capacity of sequential systems for a while already, and nowadays the need for parallel or distributed computation is extreme.

Grid computing has emerged as a cost effective alternative to the use of large supercomputers. As a computing grid, a vast collection of heterogeneous computational resources connected through wide area networks might be effectively used for high performance applications. A lot of effort is currently going towards building infrastructure that enables application developers to easily take advantage of the enormous computation power available on the Grid [7, 8, 9, 10, 11, 12].

One particular technology that has been used effectively for building Grid enabled applications is Datacutter [7, 13, 14]. This environment is based on the dataflow model, and assumes that applications are decomposed

into filters that comunicate using streams. Figure 1 illustres a Datacutter application. In the graph, the nodes represent the different filters that compose the application. These are the basic computational components of the application. The edges represent the streams, or the communication channels that connect the filters. These are unidirectional pipes over which fixed size buffers flow from one filter to another. These filters can be instantiated on several machines across the Grid and the system takes care of connecting the streams appropriately. Multiple instances of the same filter are allowed to balance the different computational requirements of different filters, and to achieve high performance.

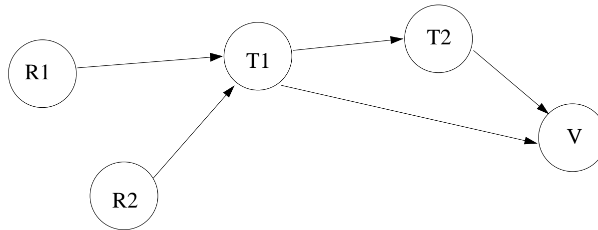

Figure 1: Datacutter application graph.

One important consideration about Datacutter is related to scheduling which, in that environment, is the process of deciding how many copies of each filter should exist to achieve high performance, and on which hosts of the grid each copy should run.

In this paper we present LPSched, a scheduling mechanism for Datacutter, based on linear programming. The idea is to provide the scheduler with basic information about the application itself and the available computational environment. Based on that information, the system will produce a datacutter schedule for efficient execution on the Grid. LPSched does that by building a linear programming problem [22, 23] based on the information it receives. Upon the solution of the problem, it generates the schedule. Our experiments have shown that LPSched is able to effectively produce Datacutter schedules that achieve good performance while being conservative about node allocation on the Grid.

This paper is organized as follows. In the next Section we described some related work on scheduling for the Grid environment. in Section 3 we describe the basics on Datacutter and the application model it implements. The following Section goes into the details of LPSched and the linear programming problem it generates to produce an efficient schedule. In Section 6 we conclude the work and present some future directions.

2 Related Work

There are several related works that target the scheduling of applications. The strategies employed in those works vary significantly, and are often specific to a given computational environment or scenario.

Partitioning an application into filters and the associated scheduling can be modeled with classical strategies [15, 16, 9, 10, 17, 18, 19], where the problem is modeled as a graph of tasks that must be scheduled (each task is a filter) and a heuristic is used to map them to the available processors. Such strategies focus just on the processing demands and frequently do not consider communication demands. They differ regarding the

decision criteria used on allocating tasks to processors. Some strategies use clustering [20, 16], others employ scheduling lists [9, 13], and there are some that employ a combination of both. In the clustering-based strategies, the problem is divided into sub-problems where the processor allocation is defined by a greedy heuristic.

The first version of Datacutter used a Filter-Copy Pipeline (FCP) scheduling [13], which is based on a cost model of the filters that compose a Datacutter application. In that model, the concept of unit of work is defined based on the division of the overall work of the application. In Datacutter, each filter performs the processing associated with each unit of work it receives and requests another unit after concluding it. The complete execution of each unit is called a work cycle. The strategy includes a controlled execution that collects data used to estimate the behavior of an actual application. The scheduler models all execution components, including network latency, communication costs, and processor capacities. The number of instances is calculated iteratively as the model receives the timings associated with each work cycle. The filter allocation is based on a greedy algorithm that priorizes the filters that demand more processing, since it allocates at most one instance per filter. The main weakness of this method is the variability of the estimations associated with work cycles as a consequence of processor availability and network connectivity. Since the availability of processors vary significantly in grids, the applicability of the model is quite restricted. Another restriction is that it does not take into consideration the network bandwidth reduction that occurs as filters are allocated to the processors, resulting in schedulings that overload connections.

3 Datacutter

In this section we describe the user-level middleware called Datacutter [21], which has been developed at the University of Maryland and the Ohio State University. Datacutter supports the filter-stream programming model, where an application executes over distributed, heterogeneous environments by allowing decomposition of application-specific data processing operations into a set of filters that interact, that is, exchange data, via streams.

The specification of a filter consists of an initialization function (init), a processing function (process), and a finalization function (finalize). A stream is an abstraction used for all filter communication, and specifies how filters are logically connected. All transfers to and from streams are through a provided buffer abstraction. A buffer represents a contiguous memory region containing useful data. Streams transfer data in fixed size buffers. Filter operations progress as a sequence of cycles, with each cycle handling a single application defined unit-of-work. A work cycle starts when the filtering service calls the filter init function, which is where any required resources such as memory or disk scratch space are pre-allocated. Next the process function is called to continually read data arriving on the input streams in buffers from the sending filters. The finalize function is called after all processing is finished for the current unit-of-work, to allow release of allocated resources such as scratch space.

The use of Datacutter relieves programmers from several low-level tasks, such as process initiation and interprocess communication, as well as it provides flexible shared memory or distributed memory parallelization

associated with its support for transparent copies. Transparent copies allow a finer level of parallelism via multiple copies of a single filter. The filter runtime system maintains the illusion of a single logical point-to-point stream for communication between a logical producer filter and a logical consumer filter. When the logical producer or logical consumer is transparently copied, the system decides for each producer which copy to send a stream buffer to. Schemes like round-robin allocation are used to achieve load balancing.

The execution of Datacutter applications is based on a configuration file, which specifies hosts, filters and their instances, and the streams. The task of scheduling an application to a cluster and/or grid means matching instances to hosts, as well as data sources, considering the connectivity among the hosts and the demands imposed by the streams. Thus, an automatic scheduling system for a Datacutter application should generate just this configuration file, determining the number of instances of each filter and where these instances execute.

4 Linear Programming Scheduler (LPSched)

In this Section we present LPSched, our scheduler based on Linear Programming. The idea behind LPSched is that it uses information about the application being scheduled and the available computational resources to create a Linear Programming Problem (LPP). Upon the solution of this problem, a Datacutter schedule can be generated.

The main idea behind our approach is to consider datacutter as running in steady state, meaning, the filters are receiving data at a given rate, processing that data, and then producing an output at a particular rate. In such framework, filters can be thought of as consumers and producers of data, and the problem becomes that of balancing the data flows coming in and out of each filter. Such problems can be modelled very conviniently using the Linear Programming methodology.

We now describe the information LPSched uses to produce its LPP. Then we present how this LPP is generated. As we shall see, the problem is very similar to the classical flow optimization problem described in the theory of Linear Programming [22, 23], in particular, we generate a Mixed Integer Linear Programming Problem, and it can be solved using a Simplex solver. Finally, we show that a schedule can be generated from the solution to the corresponding LPP.

4.1 Problem parameters

In order to produce a reasonable schedule, LPSched requires a certain knowledge. This knowledge is related to the available computational resources and to the application being scheduled.

LPSched models the platform as a set of clusters connected using wide area networks. Each cluster is composed of a number of machines that are connected throught switches. The information required for the network connections is the comunication bandwidth. For the nodes, LPSched needs some information to be able to compare different nodes on an heterogenous environment. This is provided by a number that gives the processing capability of the node as perceived by a Datacutter filter. The larger the number, the faster the filter will execute on that node.

Figure 2 shows an example of an infrastructure available for Datacutter to execute applications. There are three clusters ($C1, C2, C3$ in the picture), all connected using some WAN. The network bandwidth available between the clusters are represented by $BW[C_i, C_j]$. The other two parameters, not shown in the picture, are $BW[C_i]$, the bandwidth available within C_i and the performance indicator $P[h_i]$ for each host h_i available.

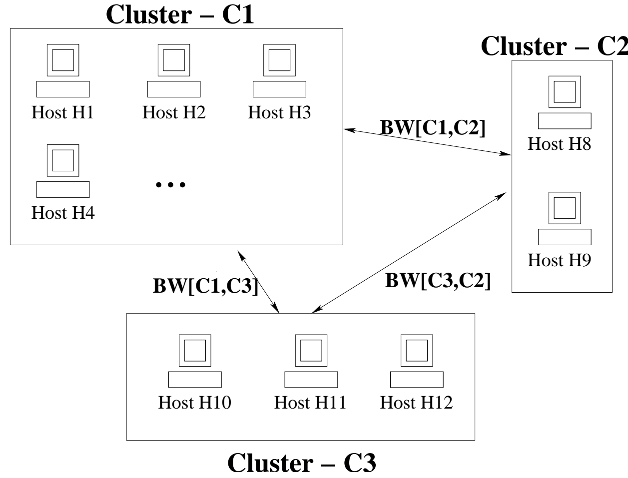

Figure 2: Available computational resources.

The applications modelled in LPSched are similar to the one presented on Figure 3. The current implementation considers only applications that can be decomposed as a pipeline of filters. The nodes on the graph represent the filters that compose the application. In our example, there are 3 filters. The first one, R, is the Read filter. This is the filter that retrieves the application data from disk and starts the data flow. Read filters are special in any Datacutter application for it is the source of the data for the remaining filters. It is specially marked and recognized by LPSched. The second filter, T, receives data chunks from the first and performs some transformation on the data. The last filter, V, is the visualization filter, which receives the transformed data and creates the output. The edges represent the streams, or the communication pipes between the filters. There are two edges shown in the picture, connecting R to T and connecting T to V.

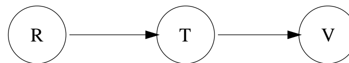

Figure 3: Sample application graph.

In addition to the application graph described above, LPSched needs some additional information about the application. First, it needs to know which nodes in the computation infrastructure contain portions of the input data. LPSched handles each group of nodes that contain a complete replica of the input data as a separate input data configuration set. Each set i is represented by two variables, $X_R[h, i]$ and $V_R[h, i]$. The first is a boolean and tells whether there should be a Read filter on host h for data source configuration i. The second variable represents the amount of data available on host h for option i as a percentage of the total input. These two variables are related and are provided by the application programmer in a configuration file. The decision of which source to use is up to LPSched.

Also, LPSched relies on information collected during a controlled execution of the application. In this controlled execution (or trial run), there is one single instance of each filter, running on a separate, unloaded host. The following information is collected for each filter in the trial run: $P[f]$, the processing power of the host which executed filter f, $V[f]$, the volume of data produced by filter f, and $T[f]$, the time spend by filter f.

This is all the information required by LPSched. Based on that it can produce the LPP automatically, as described next.

4.2 The Linear Programming Problem

As any LPP, this one requires a set of variables which are the values that need to be found by the solver. These must be such that will optimize some objective function while, at the same time, satisfying a set of restrictions posed to the problem. To create an LPP is, therefore, to create these three pieces of information (variables, objetive function, and restrictions).

4.2.1 Variables

The values that LPSched is must produce constitute the execution schedule, the information about which filter replica should be run in each node, if any. One assumption we make is that there will be only one instance of any filter running on any node. This is so that we avoid multiple filter instances competing for the node at runtime. We call these variables $vX[f, h]$ which are boolean values indicating the presence of filter f in processor h. The solution to the LPP will assign values 0 or 1 to these variables and from that information a schedule can be produced in a straight forward way.

The second set of values is related to the amount of data tranfered to and from each node. This is not actually an output for the scheduler, but a fundamental piece of information in the LPP itself. In particular, these values are used in many of the restrictions LPSched impose to the problem, since there is a limited bandwidth in and out of every node of the system. These variables, $vF[f_p, f_c, h_p, h_c]$, represent the volume of data communicated from the instance of filter f_p running on host h_p to the instance of filter f_c running on host h_c. The values are real numbers given in Megabytes per Second.

Figure 4 shows a representation of the scheduling graph of our example application assuming there are 12 hosts available to execute the application. The variables of the LPP are represented in the figure.

A third set of variables for the LPP, which is not represented in Figure 4, is related to the choices available for the input data source configuration set. A boolean variable $vOP[i]$ is used to tell whether or not configuration set i is chosen as the input data source

4.2.2 Objective function

A Linear Programming Solver will assign values to the variables of the problem in such a way as to optimize a given objective function. In the context of LPSched, an optimal schedule can be thought of in two ways.

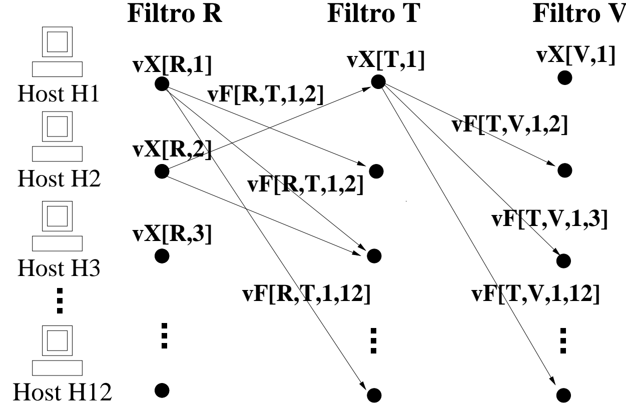

Figure 4: Scheduling graph.

First, an optimal schedule is such that the application runs as fast as possible. If we consider that filters operate on the information that flows through them, the faster the filters communicate, the faster output is produced and the computation finishes. To achieve that, LPSched should maximize the flow of data between filters. The equation below computes the total dataflow. We introduce one additional parameter here, which is a weight for cross-cluster communication, a function $W_C(C_i, C_j)$ where C_i and C_j are any two clusters ($i = j$ is a possibility). Additionally, $Cluster(h)$ is a function that returns the cluster to which host h belongs.

$$S1 = \sum_{(fp,fc)\in filters} \left(\sum_{(hp,hc)\in hosts} (W_C(Cluster(hp), Cluster(hc)) \times vF[fp, fc, hp, hc]) \right) \tag{1}$$

On the other hand, a good scheduler should consume the minimum of the available resources necessary for each application. In particular, a greedy scheduler is frowned upon on a Grid environment, for there are always multiple applications running simultaneously on the shared resources. Therefore, LPSched should be conservative about using nodes, and place filters only on as many nodes as it consider useful. The total number of processors used for any given schedule is given by the equation below. We introduce another additional weight here, W_X, which is a cost that should be paid for every additional node LPSched chooses to use.

$$S2 = \sum_{f\in filters} \left(\sum_{h\in hosts} (W_X \times vX[f, h]) \right) \tag{2}$$

The objective function adopted in LPSched is, then, a combination of Equations 1 and 2 given below.

$$maximize(S1 - S2) \tag{3}$$

The weights introduced previously in this section can be used as knobs for fine tunning of the scheduler. For instance, the user can use W_C to favor certain cluster over others. W_X can also be seen as a nice way to represent a tradeof between additional nodes and extra MB/s: an extra node should be added only if it would increase the total flow above a certain threshold.

4.2.3 Restrictions

Restrictions are used in Linear Programming to provide bounds to the values of the variables, and to relate different aspects of the problem. The restrictions are a set of inequalities on the problem variables and any

viable solution to the problem must satisfy all of them.

Several restrictions are defined by LPSched to guarantee that the schedules it produces are viable.

The first restriction is related to the **choice of input data source**. The restriction is used to guarantee that for every schedule produced, one and only one input data set is used. This situation can be modeled by the formula below.

$$\sum_{i \in options} vOP[i] = 1 \tag{4}$$

Once we guarantee that a choice is made, we have to make sure that the **Read filters are correctly placed**. That is, the Read filters on the solution should match the choice of input set made by the system. To do this, LPSched generates a set of restrictions as below.

$$\forall h \in hosts, vX[R, h] = \sum_{i \in choices} (vOP[i] \times X_R[h, i]) \tag{5}$$

LPSched can easily compute the data flow produced by a Read filter on any host h available to the application. It does that by using information from the trial run. In that run, a single Read filter that read the entire dataset ran on a single processor. The flow for that filter is given by $V[R]/T[R]$. For a different host h, the flow is $F_{out}[R, h] = V[R]/T[R] \times P[h]/P[R]$. Notice that $P[R]$ is the performance index for the host in which the filter R ran during the trial run, while $P[h]$ is the performance index for host h, as provided in the information about all hosts. With that information, LPSched produces another set of restrictions that will **create values for the variables** $vF[h]$ for the host h where the Read filters might run. These are given by the equations below. Notice the filter f in the equations. That is the filter that follows R in the pipeline. In the case of the example shown in Figure 3, it would be the filter T.

$$\forall h \in hosts, \sum_{h' \in hosts - \{h\}} vF[R, f, h, h'] = \sum_{i \in choices} (vOP[i] \times X_R[h, i] \times F_{out}[R, h]) \tag{6}$$

Next, LPSched generates a set of equations to guarantee that **at least a copy of each filter exists** in any viable schedule. The equation is given below.

$$\forall f \in filter, \sum_{h \in hosts} vX[f, h] \geq 1 \tag{7}$$

Going back to the data flows, we showed how $F_{out}[R, h]$ could be computed for filter R on host h. For any choice of input data source, multiple hosts can be used to read data. Therefore, the notion of an agregate flow exists and consists of $F_{tot}[R, i] = \sum_{h \in hosts} (X_R[h, i] \times F_{out}[R, h])$, the total aggregate data flow for filter R if option i is taken for input data source. The data produced by a filter is the data consumed on the following filter. From the trial run we know the volume of data produced on each filter. For our example application, we have from the trial run that the data produced by filter R is $V[R]$. The next filter is T, and the data produced by it is $V[T]$. So we know that filter T produces $V[T]/V[R]$ data for every byte it receives. Therefore we can extend the notion of the total aggregate flow beyond the Read filter using the trial run information as we just

described. LPSched uses that computation to provide restrictions that will create an upper bound on the data flow. These restrictions will limit the total data flow between two filters to be at most equal to that aggregate flow. The restrictions below guarantee **no more comunications than necessary** will be generated.

$$\forall edges(f_p, f_c) \in \textit{application graph}, \sum_{h_p, h_c \in hosts} vF[f_p, f_c, h_p, h_c] \leq F_{tot}[f_p, i] \quad (8)$$

We have just shown how the flow out of a filter relates to the flow into it. LPSched uses that information to produce another set of restrictions that will **guarantee that every filter will receive all the data it requires, and will also produce all the data it needs to**. The equations for these restrictions are described below.

$$\forall h \in hosts, \forall \textit{pairs of edges}(f1, f), (f, f2), \sum_{hp \in hosts} vF[f1, f, hp, h] = \frac{V[f1]}{V[f]} \times \sum_{hc \in hosts} vF[f, f2, h, hc] \quad (9)$$

For every filter f in the application, we can compute $F_{out}[f, h]$ from the information of the trial run. We have used that same computation for the Read filter (R) above to compute the maximum flow a given choice of input data source can produce. This value also represents the processing capacity for a given filter on a given node. Another set of restrictions generated by LPSched was designed to prevent **overloading of certain processors**. These restrictions will limit the flow into any filter on any host to its maximum capacity on that host. They are given by the formulas below.

$$\forall h \in hosts, \forall edges(f_p, f) \in \textit{application graph}, \sum_{h_p \in hosts} vF[f_p, f, h_p, h] \leq F_{out}[f, h] \quad (10)$$

The next set of restrictions are generated to limit the **number of filters a schedule produces**. In the equations below, we see that the second summation corresponds to the maximum computation rate available for a given filter. The first summation is the applications demand. The inequality guarantees that there will be enough compute power to consume the incoming data. Yet another fine tunning knob is provided in this equation, in the form of the parameter W_F. This value is a percentage of extra capacity that is forced on the schedule.

$$\forall edges(f_p, f_c), (1 + W_F) \times \left(\sum_{i \in options} (F_{tot}[f_p, i] \times vOP[i]) \right) \leq \sum_{h \in hosts} (vX[f_c, h] \times F_{out}[f_c, h]) \quad (11)$$

Another basic guarantee that LPSched needs to provide is that **the network bandwidth limitation is considered**. It means that the schedule will not try to push more data through a connection than that connection's capacity. There are different equations for this set of restrictions depending on characteristics of the network specified for the connection. For the switched LANs, LPSched will generate equations so that no connection between two machines on the same cluster exceeds the capacity of the switch. The equations are as follows.

$$\forall \textit{pairs of hosts}(h_p, h_c) \textit{on same cluster}, \sum_{f_p, f_c \in filters} vF[f_p, f_c, h_p, h_c] \leq BW[c] \quad (12)$$

For WANs, LPSched considers two different models: in the first one, the network connections is considered to be shared, meaning that the sum of all data flowing into that cluster need to be added up and the summation cannot be greather than the total. The other case, the connections are not shared, and therefore only the specific data flows need to be limited by the network bandwidth. The equations are very similar to Equation 12 and are omitted here.

The final set of restrictions are the ones **relating the variables vF and vX**. As mentioned earlier, the former represents the data flow between two instances of filters, while the latter indicates the existence of a copy of a given filter on a given host. These two variables are related in the sense that the existence of a flow in or out of a filter in a particular host indicates the presence of that filter running on that host. We introduce yet another parameter here, $W_{MIN}[f]$ which represents the minimum flow produced by a given filter on any host running that filter. From that parameter we compute a minimum flow, or capacity of a given filter at a given node as $F_{min}[f,h] = W_{MIM}[f] \times MIN\,(F_{out}[f,h], MAX(F_{tot}[f_c,i]))$. These variables are generated for every f and every h. With the information, the two final sets of equations are presented below.

$$vF \times vX : \forall h \in hosts, \forall edge(f_p, f_c) \in \textit{application graph}, \frac{\sum_{h_p \in hosts} vF[f_p, f_c, h_p, h]}{MAX(F_{tot}[f_c, i])} \leq vX[f_c, h] \qquad (13)$$

$$vX \times vF : \forall h \in hosts \forall edge(f_p, f_c) \in \textit{application graph}, \frac{\sum_{h_p \in hosts} vF[f_p, f_c, h_p, h]}{F_{min}[f, h]} \geq vX[f_c, h] \qquad (14)$$

With these restrictions, it is not hard to realize that if one of the variables is zero, the other one will be forced to zero, and if one of them is not zero, the other is not either. In particular, vF will be greater than $F_{min}[h,p]$ if vX is 1.

4.3 Model relaxations

The model presented is actually the most strict one. It assumes that there is enought computation power and bandwidth to serve the needs of the applications. In practice, this may not always be the case. Therefore, we implemented two relaxations of the model proposed above.

Our first relaxation is for situations in which there is enough bandwidth but we lack processing power to keep up with the data production rate of the read filters. For such cases, the strict model would not find a solution (a schedule) for the restrictions can not be met. To solve this problem, we eliminate the restrictions related to the transmitted flow. With that restriction removed, we can generate more data than the following filter can consume, therefore a schedule is possible, but it is sub-optimal.

The second relaxation is for situations in which there is not enough processing power nor bandwidth. In that case, we switch to a different algorithm and perform a scheduling based only on the total execution time of each filter in the trial run.

Both relaxations produce sub-optimal solutions, but they are required for certain practical situations and therefore they were implemented in our model. A more comprehensive description is available in [24].

4.4 Implementation

A version of LPSched has been implemented for testing and validation purposes. It is currently integrated with Datacutter original API and users can benefit from it by simply informing Datacutter that the placement of the filters will be determined by LPSched. The configuration files are all in INI file format, as are the original Datacutter configuration files. The information on the trial runs are generated by the original profiling information output by Datacutter.

One important feature of LPSched implementation is that it probes the computational environment for current loads on the machines, and therefore it is capable of generatint schedules adapted to the current status of the Grid. It does that by consulting an infrastruture built on top of the Network Weather Service (NWS) [25] which monitors current and past activity on the hosts and estimates future loads. LPSched then modifies all the information from the trial runs and the hardware capabilities to reflect the estimated load on the system.

With that information, LPSched creates a set of equations that define the LPP. These equations are generated in a format suitable to CPLEX, a sofisticated Simplex solver. LPSched then connects to CPLEX to solve the problem, retrieves the solution with the values of the variables, and produces the schedule. This last step is straight-forward, since the variables $vX[f, h]$ correspond directly to the schedule.

5 Experimental Results

In this section we discuss experimental results of the application of LPSched to several scenarios, where we are able to evaluate the actual behavior of the scheduler, and its impact in terms of both execution time and computational resource usage.

We compare LPSched to two scheduling strategies. The first strategy, which we call trivial, allocates a single processing node to each filter. The second strategy is similar, but it uses all processing nodes available, dividing them among the filters evenly. The comparison between LPSched and these strategies allows us to verify how LPSched reacts to load unbalancing among filters, in terms of creating copies for the filters that perform more computation and exploiting the available resources more effectively. We believe that these two strategies, although very simple, represent adhoc approaches that are commonly employed. In the next subsection we describe the experimental setup used in the experiments, as well as the applications evaluated. We then evaluate the effectiveness of LPSched in five scenarios.

5.1 Experimental Setup

The experiments presented in this section were performed using three geographically distributed clusters of machines, more specifically, these clusters are located at the Universidade Federal de Minas Gerais (UFMG), Brazil, at the University of Maryland, College Park, and at the Ohio Supercomputer Center. As we see in the sections that follow, these machines were organized in several configurations depending on the purpose of the evaluation. Besides Datacutter, these machines were also running NWS in order to gather information about their load and use by other applications. We used up to 19 nodes from UFMG, which are divided into 3

clusters. The first cluster comprises 8 machines (MG1 to MG8) that provide 1.4745 processing units each. The second and third clusters comprise 4 (MG9 to MG12) and 7 machines (MG13 to MG19), respectively, and each machine in those clusters provides 0.9912 processing units. All machines are connected through shared Fast Ethernet networks. We also used up to 48 machines from the University of Maryland (MD1 to MD48), each with 1.29105 processing units. Finally, we used up to 24 machines from the Ohio Supercomputer Center (OH1 to OH24) with 1.85466 processing units each. All clusters are connected internally by dedicated Fast Ethernet switches.

LPSched was used to schedule executions of two applications during the experiments. The first one is a modified version of a simple Datacutter test program, which is used for varying the load in a controlled way. The second application is the virtual microscope [1]. Next we describe each application in more detail.

The first application is a modified version of the *GridArrayAverager*, a demo application which comes as an example for Datacutter, and calculates the average of an array of numbers that is given as input. This application is divided into three filters: *ArrayReader*, *ArrayAdder*, and *ArrayAverager*. A text input file is read by filter *ArrayReader* and the numbers read are grouped in blocks and sent to filter *ArrayAdder*, which sums the numbers in each block. Each sum is sent to filter *ArrayAverager* that calculates the average of all numbers. This application is not computationally intensive and most of its cost is associated with the *ArrayReader* filter. In order to evaluate the effectiveness of our approach in a controlled environment, we modified it so that the amount of computation performed by the *ArrayAdder* filter may be amplified as needed, demanding load balancing measures.

The second application used in the experiments is the virtual microscope [1, 13], a popular application in the Datacutter domain. It allows researchers to view selected portions of a large, highly detailed image with different resolutions, much like an actual microscope does. It is implemented using 5 Datacutter filters. The first filter is the *Reader*, which is responsible for reading the compressed image requested by the user. The compressed data are sent to the *Decompress* filter that restores the original image data. The *Clip* filter then selects the portion of the image that the user wants to see and is followed by the *Zoom* filter, which focus the image to the desired granularity. Finally, the *View* filter generates the final image that is presented to the user.

During the experiments described in the next sections, we evaluate two popular metrics: execution time and resource utilization. The results presented are the average over three executions. There were not significant variations among the executions. As mentioned, we evaluate the effectiveness of LPSched regarding five criteria in the following sections: load balancing, multiprogramming, variable I/O throughput, intra- and inter-cluster connectivity.

5.2 Load Balancing

The first set of experiments demonstrates the effectiveness of LPSChed in determining the proper amount of computational resources for applications that are composed of filters that impose diverse processing demands. We employed the modified GridArrayAverager and varied the demand posed by the *Array Adder*. We used seventeen of the nineteen machines available in UFMG. In particular, we did not use two of the faster machines,

MG7 and MG8.

In the first experiment, the *ArrayAdder* demands two times more processing than the other filters (being denoted by 1 x 2 x 1). We varied the amount of input data from 2 to 32 MB and evaluated each of the aforementioned scheduling strategies (Trivial, LPSched, and Total). The results are shown in Figure 5.

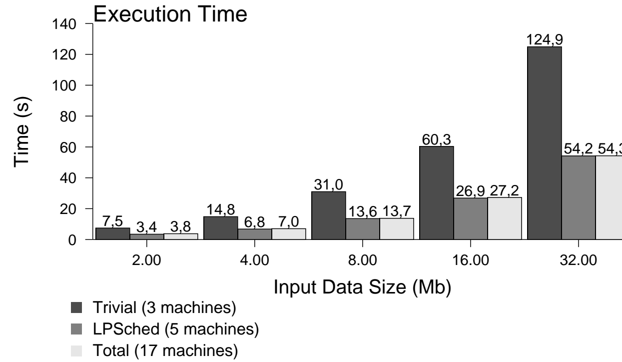

Figure 5: Execution Time of the Modified GridArrayAverager — 1 x 2 x 1

The scheduling generated by LPSched used just 5 of the 17 available processors, in particular the most powerful machines. The first and last filters were assigned to a single machine each, while the *ArrayAdder* was executed by three machines. We can easily observe that LPSched outperforms the Trivial scheduling significantly, being more than 2 times faster. Further, it provides slightly better performance than the Total configuration, using less than one third of the processors. In this case, the gains are explained by the smaller amount of traffic generated by the LPSched configuration and its choice for faster machines, reducing the demand and thus the amount of communication necessary.

We then performed a second set of experiments employing the three scheduling strategies where the *ArrayAdder* demands five times more processing than the other filters (1 x 5 x 1). The results of these experiments are presented in Figure 6. LPSched allocated seven machines for these executions, more specifically MG1 to MG6, and MG9. The first observation is that the gains of LPSched over the Trivial scheduling are even greater, running almost 5 times faster. Further, regardless of the higher imbalance, the overall execution times are comparable to the executions where the imbalance level is 2. LPSched was also able to recognize the filters that demand more processing, and allocate the most powerful processors to the *ArrayAdder* and *ArrayReader* tasks, while the *ArrayAverager*, which is less computationally intensive, runs on MG9, which is slower. In summary, the use of LPSched allowed to allocate the proper number of machines and to take into consideration the performance of the machines to achieve high efficiency.

5.3 Multiprogramming

The goal of this experiment is to evaluate the use of LPSched in multiprogrammed environments. We used all machines from UFMG and placed an external load of 50% in some of the machines (MG4, MG5, MG6, MG7, MG8, MG10, MG13, and MG14), being able to measure it using NWS. We them compared the scheduling

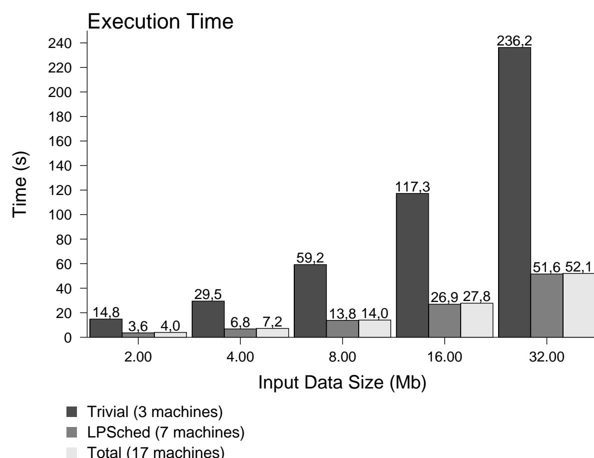

Figure 6: Execution time of the Modifi ed GridArrayAverager — 1 x 5 x 1

generated by LPSched to the Total scheduling for a modifi ed *GridArrayAverager* with a load imbalance factor of 7, that is, the *ArrayAdder* fi lter demands 7 times more processing than the other fi lters. LPSched used 12 of the 19 processors available, and sustained a better performance than the Total scheduling for all input sizes, as can be seen in Figure 7. The performance gains may be explained by the fact that LPSched allocated load proportional to the availability of each machine, while the Total scheduling does it in a round-robin fashion. In particular, it used the machines MG1 for the fi lter *ArrayReader*, MG2, MG3, MG9, MG10, MG11, MG12, MG14, MG15, MG16, and MG17 for the fi lter *ArrayAdder*, and MG18 for the fi lter *ArrayAverager*, demonstrating that LPSched is able to generate effi cient schedules in multiprogrammed environments.

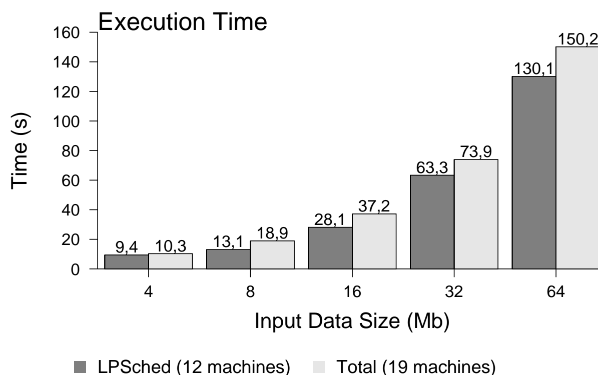

Figure 7: Execution time of the Modifi ed Grid Array Averager using the NWS — 1 x 7 x 1

5.4 Variable I/O Throughput

The experiments presented in this section evaluate the ability of LPSched to generate good assignments considering the I/O confi guration and data distribution of the input data. In these experiments we employed all 19 machines from UFMG, and the modifi ed *GridArrayAverager* with an imbalance factor of 5, using an input data fi le of 32 MB. Regarding input data source, we used two different types of machines. The fi rst type is

associated with the machines MG1 to MG8, while the second type is associated with the machines MG13 to MG19. The first type is 3 times more powerful than the second type when we consider their I/O speeds and processing power.

In the first experiment, LPSched chose between two input data source configurations: (1) MG1 (providing 32MB), and (2) MG13 and MG14 (each providing 16MB). LPSched chose the first option, as expected, resulting in a overall execution time of 53.2 seconds. In our second experiment, we changed the second configuration, dividing the data evenly among the machines MG13 through MG16 so that each machine was responsible for 8MB of input data. In this case, LPSched chose the later, resulting in an execution time of 41.7 seconds, an improvement of 22%.

We then evaluated the ability of LPSched to handle both multiprogramming and I/O throughput. We repeated the first experiment, but placed an external load of 50% in MG1. In this case, LPSched chose the second option (MG13 and MG14), resulting in an execution time of 81.2 seconds. We also repeated the second experiment, but placing an external load of 50% in each of the machines MG13 to MG16C, when MG1 was chosen by LPSched. These experiments showed not only the ability of LPSched to take into consideration the I/O throughput of machines, but also to generate good schedules considering tradeoffs in more than one dimension, in particular multiprogramming and I/O throughput.

5.5 Intra-cluster Connectivity

This set of experiments demonstrates the ability of LPSched to handle connectivity restrictions among machines that belong to a cluster. We employed the machines from the University of Maryland (MD1 to MD48) and the virtual microscope application, which is remarkable because of its I/O demands. The *Decompress* filter generates a large amount of data, while the filters *Clip*, *Zoom*, and *View* perform a small amount of processing compared to the communication time, making it a good application for sake of evaluating the impact of communication constraints. We employed the three scheduling strategies and all data is located in MD1, which also runs the *Reader* filter in all cases. The Trivial scheduling chose just 5 machines, one for each filter. The Total scheduling allocates machines according to the expected load for each filter, that is, 35 machines (MD2 to MD36) run the *Decompress* filter, 6 machines (MD37 to MD42) run the *Clip* filter, 5 machines (MD43 to MD47) run the *Zoom* filter, and one machine (MD48) runs the *View* filter. Finally, LPSched allocated 45 machines, using 31 machines for the *Decompress* filter, 7 machines for the *Clip* filter, 5 machines for the *Zoom* filter, and 1 machine for the *View Filter*.

The execution times for all experiments are shown in Figure 8. As expected, both Total and LPSched resulted in better performance than Trivial, and LPSched outperformed Total, improving the execution time up to 25% regardless of the fact it saved 3 machines. LPSched used a flow relaxation model, which takes into account the flow of data per filter.

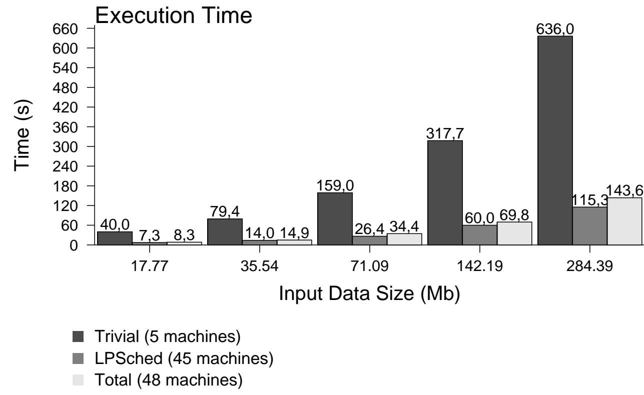

Figure 8: Intra-cluster network connectivity restrictions — Virtual Microscope Application

5.6 Inter-cluster Connectivity

The goal of the last set of experiments is to demonstrate the ability of LPSched to determine good schedules considering the connectivity inter clusters of machines. For this evaluation we use again the virtual microscope because of the amount of communication it performs. We used the clusters from the University of Maryland and from the Ohio Supercomputer Center. Again, we employed the three scheduling strategies: Trivial, Total, and LPSched. The trivial scheduling allocates one machine per fi lter, all of them in the MD cluster. The total scheduling allocates MD1 to the *Reader* fi lter, 47 MD machines (MD2 to MD48) and 8 OH machines (OH1 to OH8) to the *Decompress* fi lter, 10 OH machines (OH9 to OH18) to the *Clip* fi lter, 5 OH machines (OH19 to OH23) to the Zoom fi lter, and the fi lter *View* is allocated to OH24. We also set the minimum machine utilization to 70% and kept all other parameters from the intra-cluster experiment.

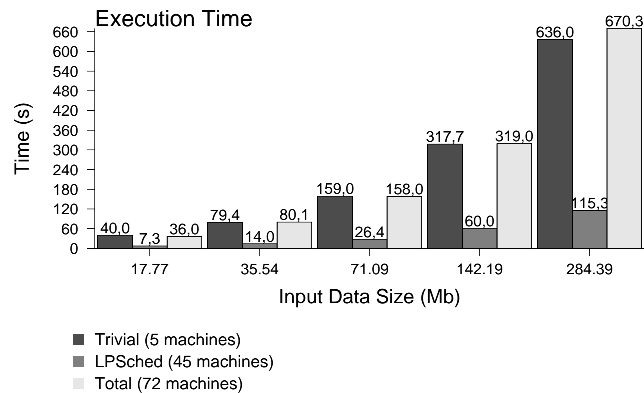

Figure 9: Inter-cluster network connectivity restrictions — Virtual Microscope Application

The results are presented in Figure 9 where we can see that the Total scheduling is the worst option for larger input data, as a consequence of the utilization of the OH machines, which increased the delays due to the inter-cluster traffi c. LPSched indicated the same allocation of the intra-cluster experiment, since it considered that it was not worth communicating through the shared link between the two clusters. The results show that the

use of all machines is as bad as using just one machine per filter, that is, the machine contention of the Trivial scheduling results in the same performance of a shared long-distance connection between a large number of machines. Further, LPSched was able to recognize such problem and decided not to use all computational resources available. In this case, the placement of the initial data and the minimum utilization restriction were the reasons behind the utilization of the MD machines.

6 Conclusions and future work

The advances in many fields of science and engineering are bringing about an incredible new demand for storage and processing power. Distributed environments like Clusters and Grids have emerged as serious platforms to provide the capabilities demanded by these new applications. Datacutter is a run-time framework that enables applications to run on Grid environments. It is based on a dataflow model in which the application is decomposed into filters that comunicate throught streams. A critical portion of achieving high performance in Datacutter applications is by making good choices about how many copies of each filter to create and where to run them.

We presented LPSched, a scheduler that is capable of generating good schedules for Datacutter applications. LPSched is based on linear programming methodology which has proven very convenient to model Datacutter scheduling problem. From our experiments, we learned many lessons: the optimum schedule does not necessarily need all available hosts. Because the data input bandwidth limits the overall performance, all we need is enough resources to keep up with that bandwidth. We have seen that in some cases, having extra copies can actually hurt the overall system performance.

We also learned that a good forecast for resource availability is paramount to a correct schedule. If critical filters end up on unavailable, or overloaded hosts, significant performance degradation can happen. Correct estimations of the network bandwidths are also critical. Not considering this factor can severely limit the system performance and increase waste of resources.

For the applications, balancing the communication and computation for the filters is crucial. In particular, there should be enough computation on each filter to justify the cost of communication (latency). Otherwise, the applications may now scale well when the number of hosts increase. Programmers need to be careful when decomposing the application into its component filters.

With regard to LPSched, it was shown to be a good strategy for scheduling Datacutter applications. A good compromise on performance and racionalization of resources can be seen from the results of our extensive experimentation. When comparing LPSched to other schedulers in the literature, we see that it presents some improvements on to of the classic scheduling algorithms. While the latter is based on heuristics that cannot guarantee optimality, while LPSched is based on Linear Programming and it does generate optimum schedules that satisfy all the restrictions of the system.

Two things jump to mind to extend the current work:

1. Consider more general application graphs. In particular, graphs with loops and different fan-ins and

fan-outs.

2. While LPSched produces what the optimal dataflow should be across fi lters, Datacutter does not have any mechanisms to obey that determination. This can lead to suboptimal executions of optimal plans.

References

[1] A. Afework, M. Beynon, F. Bustamante, A. Demarzo, R. Ferreira, R. Miller, M. Silberman, J. Saltz, A. Sussman, and H. Tsang. Digital dynamic telepathology - the virtual microscope. In *Proceedings of the 1998 AMIA Annual Fall Symposium. American Medical Informatics Association*, Nov 1998.

[2] Microsoft TerraServer. Microsoft corp. http://www.terraserver.microsoft.com, 1998.

[3] Veloso A. and Meira W. New parallel algorithms for frequent itemset mining in very large databases. In *Proceedings of the 15th Symposium on Computer Architecture and High Performance Computing, So Paulo, SP*, 2003.

[4] G. Patnaik, K. Kailasnath, and E.S. Oran. Effect of gravity on flame instabilities in premixed gases. *AIAA Journal*, 29(12):2141–8, Dec 1991.

[5] Kwan-Liu Ma and Z.C. Zheng. 3d visualization of unsteady 2d airplane wake vortices. In *Proceedings of Visualization'94*, pages 124–131, Oct 1994.

[6] T. Tanaka. Confi gurations of the solar wind flow and magnetic fi eld around the planets with no magnetic fi eld: calculation by a new mhd. *Jounal of GeophysicalResearch*, 98(A10):17251–62, October 1993.

[7] DataCutter Project. Datacuttermiddleware for fi ltering large archival scientifi c datasets in a grid environment.
http:/www.datacutter.org, 2004.

[8] Francine D. Berman, Rich Wolski, Silvia Figueira, Jennifer Schopf, and Gary Shao. Application-level scheduling on distributed heterogeneous networks. In *Proceedings of the 1996 ACM/IEEE conference on Supercomputing (CDROM)*, page 39. ACM Press, 1996.

[9] H. Topcuoglu, S. Hariri, and M. Wu. Task scheduling algorithms for heterogeneous processors. In *Proceedings of the 8th Heterogeneous Computing Workshop - IEEE Computer Society Press*, pages 3–14, April 1999.

[10] J. Weissman and X. Zhao. Run-time support for scheduling parallel applications in heterogeneous NOWS. In *Proceedings of the 6th High Performance Distributed Computing Conference*, aug 1997.

[11] Douglas Thain, Todd Tannenbaum, and Miron Livny. Distributed computing in practice: The condor experience. *Concurrency and Computation: Practice and Experience*, 2004.

[12] Casanova H., Legrand A., Dmitrii Zagorodnov D. and Berman F. Heuristics for scheduling parameter sweep applications in grid environments. In *Proceedings of the 9th Heterogeneous Computing workshop (HCW'2000)*, pages 349–363, 2000.

[13] M. Beynon. Supporting data intensive applications in a heterogeneous environment. In *Phd Thesis 2001 University of Maryland at College Park*, 2001.

[14] Nazareno Andrade, Walfredo Cirne, Francisco Brasileiro, and Paulo Roisenberg. Ourgrid: An approach to easily assemble grids with equitable resource sharing. In *Proceedings of the 9th Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP)*, June 2003.

[15] Yang Y. And Casanova H. UMR a multi-round algorithm for scheduling divisible workloads. In *Proceedings of the International Parallel and Distributed processing Symposium (IPDPS'03)*, Nice, France, april 2003.

[16] M. M. Eshaghian and Y. C. Wu. Mapping heterogeneous task graphs onto heterogeneous system graphs. In *Proceedings of the 6th Heterogeneous Computing Workshop - IEEE Computer Society*, pages 147–160, April 1997.

[17] Su A., Casanova H. and Berman F. Utilizing DAG Scheduling Algorithms for Entity-Level Simulations. In *Proceedings of HPC 2002*, San Diego, CA, april 2002.

[18] Asim YarKhan and Jack Dongarra. Experiments with scheduling using simulated annealing in a grid environment. In *Proceedings of the Third International Workshop on Grid Computing*, pages 232–242. Springer-Verlag, 2002.

[19] Casanova H. Dail H. and Berman F. A modular scheduling approach for grid application development environments. Technical Report CS2002-0708, UCSD CSE, 2002. Submitted to Journal of Parallel and Distributed Computing.

[20] Taura K. and Chien A. A heuristic algorithm for mapping communicating tasks on heterogeneous resources. In *Proceedings of the Heterogeneous Computing Workshop 2000*, University of California, 2000.

[21] M. Beynon, C. Chang, U. atalyrek, T. Kur, A. Sussman, H. Andrade, R. Ferreira, and J. Saltz. Processing large-scale multi-dimensional data in parallel and distributed environments. *Parallel Computing*, 28(5):827–859, 2002.

[22] Jarvis J.J. Bazaraa M. and Sherali H.D. Linear programming and network flows. Book-Second Edition, John Wiley and Sons, 1990.

[23] Oliveira A.A.F. Bregalda P.F. and Bornstein C.T. Introduo programao linear. Terceira Edio, Ed. Campus, 1988.

[24] Luiz Thomaz do Nascimento. Escalonamento de aplicações de fluxo de dados. Master's thesis, Universidade Federal de Minas Gerais, 2004. (in Portuguese).

[25] NWS. Network weather service.
http://nws.cs.ucsb.edu, 2004.