Scheduling Parallel Applications in Distributed Networks

Jon B. Weissman and Xin Zhao

Division of Computer Science
University of Texas at San Antonio
San Antonio, TX 78249
(jon@cs.utsa.edu)

Abstract

Prophet is a run-time scheduling system designed to support the efficient execution of parallel applications written in the Mentat programming language [7]. *Prior results demonstrated that SPMD applications could be scheduled automatically in a ethernet-based local-area workstation network with good performance* [14][16]. *This paper describes our recent efforts to extend Prophet along several dimensions, improved overhead control, greater resource sharing, greater resource heterogeneity, wide-area scheduling, and new application types. We show that both SPMD and task parallel applications can be scheduled effectively in a shared heterogeneous LAN environment containing ethernet and ATM networks by exploiting the application structure and dynamic run-time information.*[1]

1.0 Introduction

Scheduling parallel applications in distributed networks is a complex problem that requires a coordinated effort between the scheduling system and the applications programmer. The complexity arises out of three distinguishing features of the network environment; the distribution of resources across multiple networks, the heterogeneity of network resources, and the sharing of network resources.

1. This work was partially funded by NSF ASC-9625000.

The dynamic nature of the network environment suggests that a compiler-only solution to the scheduling problem is unlikely to be adequate. Similarly, run-time system schedulers and operating system schedulers often make assumptions inappropriate for parallel applications. For example, adaptive load sharing, a common system-level scheduling technique, assumes tasks are uniform and independent [4]. These approaches typically do not exploit the structure of parallel applications or the heterogeneity of system resources. For example, individual application tasks are often scheduled without regard for their communication relationships which can lead to high communication overhead. A third option is to leave scheduling in the hands of the programmer providing access to a low-level scheduling API. We believe this option will restrict network-based parallel processing to the expert parallel programmer. Instead, a middle-ground solution is to provide automated run-time scheduling with programmer-provided application information. A system called Prophet has been developed with these objectives.

Prophet supports the execution of SPMD applications in heterogeneous workstation networks [14][16]. It selects the best CPU and network resources to allocate to the application, decomposes the application to use to the selected resources, and exploits the structure of SPMD applications to produce high-quality scheduling decisions. One of the unique features of Prophet is the use of run-time granularity information to select the best number of processors to apply to the application. Prophet scheduling is cost-driven where the cost models are constructed from programmer-provided application information. Prophet has been integrated into Mentat [7], an object-oriented parallel processing system developed at the University of Virginia, and significantly improved the scheduling quality for SPMD Mentat programs over the native Mentat scheduler in an ethernet network environment [15].

Previously, Prophet was limited to SPMD applications in local-area ethernet networks, and now supports ATM and wide-area networks, and other application types including task parallel pipelines

(PP). Although Prophet overhead was manageable for most applications, there was no way to guarantee acceptable overhead. The system is now *overhead-aware* and now provides better control of run-time scheduling overhead. Prophet also assumed limited sharing of CPUs and networks and now supports highly-shared networks. Experimental results indicate that SPMD and PP applications can be scheduled very effectively by Prophet to produce reduced completion time with small overhead. The results were obtained in a highly-shared heterogeneous LAN environment containing ethernet and ATM subnetworks. Prophet utilizes dynamic system information about the level of CPU and network sharing to produce high-quality scheduling decisions. The results are very encouraging and provide further evidence that run-time scheduling support can be effectively provided to a large and useful class of parallel applications in a heterogeneous network environment.

Several research projects are investigating the problem of scheduling parallel applications in shared heterogeneous networks. The DataParallel C system provides scheduling and dynamic load balancing for data parallel applications [10]. The advantage of this approach is that the programmer does not need to insert any scheduling directives or provide any application information. However, the DataParallel C system is limited to scheduling regular SPMD applications written in DataParallel C and is not broadly applicable. Another interesting approach is Apples, an application-level scheduling system [1]. The Apples approach recognizes that applications require special-purpose tuned schedulers to achieve the best possible performance. In Apples, a programmer-provided scheduling agent is responsible for scheduling and is tightly-coupled to the application. While the Apples approach is very general, it requires that the programmer write the scheduling agent, which may be a complex task. Prophet provides a scheduling capability that is decoupled from the application and automated for the user as in DataParallel C, but is not limited to a particular application type. Prophet is also *application-aware* to achieve the best performance for the application.

This paper is organized as follows. Section 2.0 reviews the Prophet architecture and its key features. Section 3.0 describes the extensions to Prophet. Section 4.0 presents the performance results obtained by scheduling applications in a heterogeneous LAN environment using Prophet. Section 5.0 presents a summary and future work.

2.0 Prophet Overview

Prophet utilizes a cost-driven method for scheduling SPMD applications in heterogeneous workstation networks. Scheduling SPMD applications consists of two parts, partitioning and placement (Figure 1). In partitioning, the application is decomposed across a set of heterogeneous processors. The number of processors selected depends on the computation granularity — an application with a large computation granularity may benefit by a large number of processors relative to a smaller one. Processor selection also depends on the granularity supported by the network resources. A faster network (e.g., ATM vs. ethernet) may allow greater parallelism to be exploited. In placement, a task will be placed on each selected processor and assigned a piece of the data domain. Tasks are assigned to specific processors to reduce communication overhead

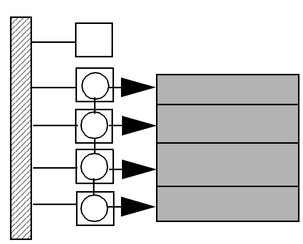


Figure 1: SPMD scheduling. The application is decomposed into a set of data regions (shaded rectangles) each assigned to a task (circle) which is placed on a processor (square). In this example, the tasks are arranged in a 1-D communication topology.

induced by routing and contention. In the SPMD model, the tasks are identical and are parameterized to compute on their data region. The data domain is decomposed to give load balance — tasks on heterogeneous processors are given a portion of the data domain proportional to the power of the processor.

Prophet exploits information about the SPMD application and the network resources in order to make scheduling decisions (Figure 2). Network resources are organized into *processor clusters*. Processor clusters contain computers on separate subnetworks with communication bandwidth shared within a cluster, but not between clusters (Figure 3). Prophet uses the following cluster information to help guide its scheduling decisions: cluster latency and bandwidth, topology-specific communication cost functions, number and type of processors, and aggregate processing power. All parameters reflect peak cluster performance.

Information about the SPMD application is provided by application *callbacks*. These are functions that provide important information about the computation and communication structure of the application (Figure 4). Callbacks for a five-point stencil application as depicted in Figure 1 are given in Figure 4b (the *arch_cost* callback is omitted, and *cycles* is undefined). The computation callbacks are based on the *PDU* (primitive data unit), which is the smallest piece of the data domain within the application. For example, the *PDU* might be a row of a grid as in the stencil

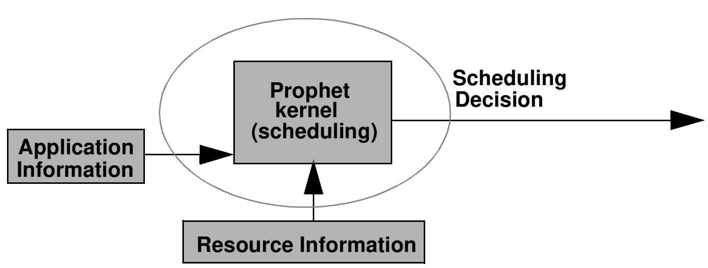


Figure 2: Prophet system architecture

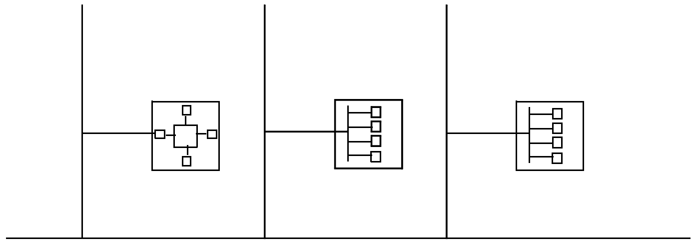


Figure 3: Network organization. A environment containing 3 processor clusters, two connected via ethernet and the other via ATM.

application. A full discussion of callbacks can be found in [14]. The scheduling process will assign *PDU*s to each task proportional to the power of the processor to which it is assigned.

Prophet explores a set of candidate processor configurations to apply to the application in order to minimize completion time. The details of this process can be found in [14]. An idealized view of the relationship between processor selection and completion time, T_{CT}, is depicted in Figure 5. The picture indicates that there are several scheduling regions. In region A, additional parallelism may be exploited by adding processors until a point of diminishing returns is reached. After this point, adding more processors will increase communication costs greater than the decrease in computation costs. In practice this function is more likely to be multimodal with several regions corresponding to A and B, especially if the processors and interconnection network

topology : 1-D, tree, ring, broadcast
comm_complexity : avg message size in bytes
numPDUs : num of data elements in problem
comp_complexity : # of executed instructions per PDU
arch_cost : arch-specific execution cost per instruction for the PDU (usec/instruction)
cycles: number of iterations if known

(a)

topology ⇒ 1-D
comm_complexity ⇒ 4N
numPDUs ⇒ N
comp_complexity ⇒ 5N
arch_cost ⇒ Sparc5: = .00015
⇒ Ultrasparc: = .00015
⇒ Sparc 2: = .0005

(b)

Figure 4: Callbacks. (a) lists the callback functions. (b) shows the callbacks for the five-point stencil shown in Figure 1.

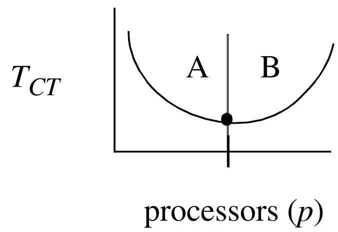


Figure 5: Finding the optimal scheduling point.

are heterogeneous. The objective of Prophet is find a scheduling point close to the minimum without expending significant run-time overhead. A guarantee to locate this minimum would require that Prophet explore an exponential number of processor configurations which is not practical. Instead, Prophet uses a heuristic which in simulation (and in practice) has yielded excellent results. The basis of this heuristic is the construction of a set of cost functions used to predict application completion time T_{CT} for each candidate schedule[2]:

$$T_{CT}[SPMD] = cycles \cdot (\overline{T_{comm}} + \overline{T_{comp}}) \qquad \text{(Eq.1)}$$

$$\overline{T_{comp}} = \frac{comp\_complexity \cdot \overline{arch\_cost} \cdot numPDUs}{p} \qquad \text{(Eq.2)}$$

$$\overline{T_{comm}}[SPMD] = (c_1 \cdot p) + (c_2 \cdot p \cdot comm\_complexity) \qquad \text{(Eq.3)}$$

Different processor configurations will have different T_{CT} values — the algorithm searches for a configuration that is predicted to give a minimum T_{CT}. $\overline{T_{comp}}$ is the average computation time spent by a processor or task in one cycle or iteration of execution. It is based on $\overline{arch\_cost}$, the average cost of executing a *PDU* over all the selected processors, p. $\overline{T_{comm}}$ is the average communication time spent by a processor or task in one iteration of execution. The communication function shown is for SPMD applications on ethernet which is linear in the number of processors, p, and message size, given by *comm_complexity*. When multiple processor clusters are used,

2. The algorithm tries to minimize the sum of T_{comm} and T_{comp}, hence the number of cycles is not required.

the communication cost functions are combined appropriately [14]. The form of (Eq.3) is typical for SPMD applications with simultaneous communication among the tasks, thus, p processors, will contend for the network concurrently. The constants c_1 and c_2 depend on the application communication topology and are determined off-line — c_1 is a latency-dependent constant and c_2 is a bandwidth-dependent constant.

In addition, the scheduling process determines an assignment of *PDU*s to the selected processors where A_i is the number of *PDU*s assigned to the task on processor i and w_i is the relative processor weight (for this application):

$$A_i = \sum_{j=1}^{P} w_i \cdot NumPDUs \qquad \text{(Eq.4)}$$

$$w_i = \frac{1}{arch\_cost[i] \cdot \sum_{j=1}^{p} \frac{1}{arch\_cost[j]}} \qquad \text{(Eq.5)}$$

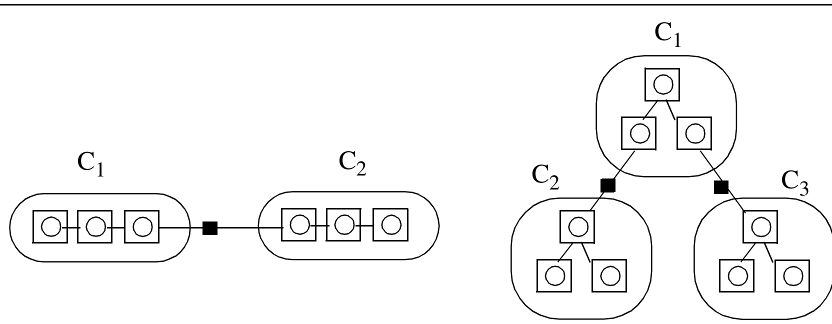
(Eq. 4) has the property that faster processors will receive a greater share of the data domain and processors in the same cluster will receive an equal share. All equations are constructed using callback information.

Another step of the scheduling process is the placement of tasks to processors. Since T_{comm} is dependent on task placement, the placement algorithm is run when evaluating each configuration explored by Prophet. The placement algorithm exploits information about the application topology and the network topology to assign tasks to processors in a communication efficient manner. Reducing communication costs is achieved by (1) maintaining communication locality (i.e., avoiding router crossings) and (2) effectively exploiting communication bandwidth within clusters. The former is achieved by inter-cluster placement and the objective is to minimize communication costs between clusters. The latter is achieved by intra-cluster placement and the objective is to minimize communication costs within clusters. Intra-cluster placement is also known as mapping or embedding and has been widely studied. Algorithms for both forms of placement are topology-specific. Inter-cluster placement depends on the communication topology. In Figure 6, we present several inter-cluster placement strategies for the 1-D, ring, and tree topologies across several clusters (the black boxes are routers). Notice that the number of router crossings or communication hops are minimized.

3.0 Prophet Extensions

3.1 Overhead Control

Since scheduling is a run-time process, it is crucial that run-time overhead be as small as possible. Prophet overhead is now controlled by exploiting knowledge of the application. Prophet will spend more time scheduling a long-running application that can better mask the additional scheduling overhead. The Prophet overhead for exploring a single processor configuration was benchmarked off-line for each architecture type and the average, $T_{Prophet}$, is available to the scheduling algorithms. This measure does not include the cost to acquire dynamic state information from each cluster discussed in Section 3.3. The amount of Prophet overhead experienced by



(a) 1-D and ring (b) tree

Figure 6: Inter-cluster placement

the application is constrained to be proportional to the expected sequential run-time of the application averaged over all processor types, T_{seq}. Currently, Prophet overhead must fall within 10% of the estimated sequential run-time of the application. Prophet overhead is controlled by limiting the number of processor configurations, *num_configs*, explored during scheduling:

$$num\_configs = \frac{T_{seq} \cdot 0.1}{T_{Prophet}} \tag{Eq.6}$$

$$T_{seq} = comp\_complexity \cdot \overline{arch\_cost} \cdot numPDUs \tag{Eq.7}$$

Both T_{seq} and *num_configs* are easily computed using the callbacks. For applications that run longer finding a high quality schedule is more important, so Prophet will spend greater overhead to explore more processor configurations. For example, while the user might prefer a schedule that takes 1 minute vs. 2 minutes, they would likely demand a schedule that takes 1 day vs. 2 days. It might also be desirable to adjust the overhead percentage as a function of T_{seq} or to allow the end-user to control this parameter. This option will be explored in the future.

3.2 Wide-area Scheduling

Prophet has been extended to run a wide-area environment to exploit opportunities for better performance. Other projects have similar objectives [1][6][8][11]. The heterogeneity, ubiquity, and availability of Internet resources can be exploited to locate the best *intra-site* resources for user applications. Running applications across multiple sites, known as *metacomputing*, requires high-speed networks running at OC-3 or DS-3 speeds. Small-scale OC-3 and DS-3 speed wide-area networks are emerging to support metacomputing, but Internet-wide metacomputing is not practical for most applications given current Internet technology. This opportunity will be explored in the near future.

Executing an application remotely using Prophet requires that the cost parameters (i.e., the callbacks) and application binaries and files be available to all remote sites. This information is encapsulated by a Mentat class called a *Job_class* (Figure 7). The programmer writes a simple front-end program which creates a *Job_class* object and submits it to the system for execution. More details about this process can be found in [17].

The wide-area network contains a set of sites with each site organized internally as in (Figure 3). Each site runs a *scheduling manager* (*SM*) and a *local scheduler* (*LS*) process. The *SM*'s run the wide-area scheduling algorithm. The *SM* interfaces to the site *LS* and to the other *SM*'s. The *LS* is responsible for managing the local site resources in a manner transparent to the *SM*. It is the *LS* that decides what resources the site will make available for wide-area scheduling at any point in time, thus providing site autonomy. The *LS* contains two components. One component is an interface to the *SM* and the other is an interface to a site-specific scheduling system, e.g. Prophet (Figure 8). A system called Gallop that allows other local schedulers such as Condor or NQS to be used is being developed [17].

```
persistent mentat class Job_class {
public:
  // *****************************
  // Execution parameters
  // *****************************
  arg_list *args; // includes binaries, files
  main_prog_class main_prog; // the main program
  arg_list *get_arg_list ();

  // *****************************
  // The callbacks
  // *****************************
  domain *job_domain;
  domain *get_job_domain ();
};
```

Figure 7: Job_class specification

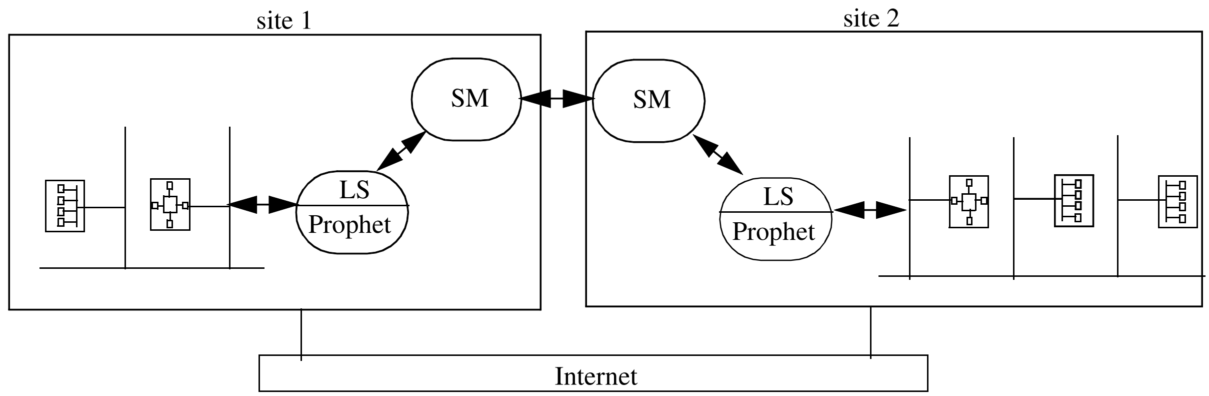


Figure 8: Wide-area system organization

The wide-area scheduling process contains a global and local phase. The global phase is initiated by the arrival of a scheduling request to a local site *SM*. The *SM* is passed the `Job_class` object to initiate scheduling of the application. The wide-area scheduling algorithm is a distributed algorithm that has two parts, the local site *SM* component and the remote site *SM* component. When an application request first arrives at a local site *SM*, a set of candidate sites are selected (*including* the local site) and the `Job_class` object is sent to each remote site *SM* for *bidding*. Selecting the candidate sites is done in the following manner. Sites "closest" to the local site in terms of communication performance are given the greatest priority since files, binaries, results, and scheduling control messages may have to be transmitted between the local and a selected remote site. Communication closeness is determined by probing the network at regular intervals. For most applications, it is too expensive to consider all available sites. An estimate of the scheduling overhead is used to constrain the number of sites under consideration. This overhead has two terms, protocol overhead, and file transfer overhead in the event that a remote site is selected. The overhead estimate is computed using the measured network bandwidth and latency to each site (for the wide-area scheduling protocol itself and for file transfer), and the local scheduling overhead ($T_{Prophet}$). Fortunately, the local scheduling overhead is paid once as the sites run this algorithm in parallel. But the communication overhead for the wide-area scheduling protocol and file transfer dominates.

The number of sites selected for bidding is constrained such that the estimated scheduling overhead falls within 10% of the best execution time that can be achieved locally. The best local execution time is obtained by having the *LS* (i.e., Prophet) generate the best schedule for the local site. For small applications, perhaps no remote sites will be considered, but for larger applications that run longer, a larger number of remote sites will be selected.

Once the candidate set of sites is determined, the local *SM* contacts each selected site *SM* to start the scheduling process. Each *SM* then contacts its *LS* to get a list of machines and when they are available. The remote *SM* then runs a local scheduling algorithm to search this set of machines to evaluate the possible schedules. In the current implementation, the *SM* invokes Prophet to evaluate the schedules. The best schedule and projected completion time are returned to the local site *SM*. Each *SM* locks this schedule until the scheduling transaction is complete. The best schedule has the smallest predicted completion time. When the local *SM* receives all of the bids, it selects the best site, initiates the scheduling of the application on that site, and informs all of the other remote sites of its decision. The chosen site *SM* passes the application (i.e., the `Job_class` object) to the local *LS* for execution. At this point, the other remote *SM*'s can release the lock on their best candidate schedule. The details of this algorithm may be found in [17]. The interactions between the various system components are depicted in Figure 9 (site SM_2 is selected to run the application).

3.3 Highly-Shared Networks and Dynamic Resource Monitoring

Previously, the Prophet scheduling process made limited use of dynamic resource information —dynamic processor load information was used only to mark processors available (if lightly-

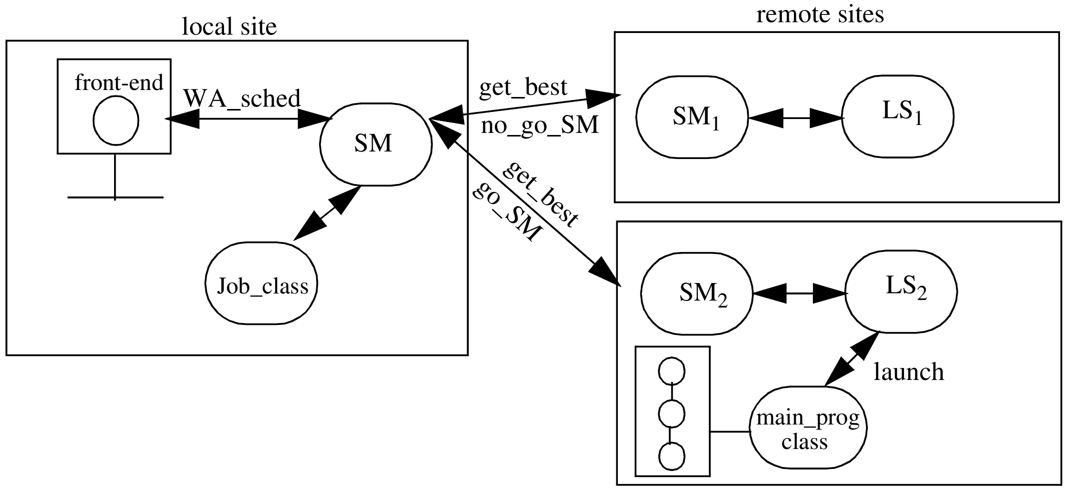


Figure 9: Wide-area scheduling and execution

loaded) or unavailable (if highly-loaded), and no dynamic network information such as available bandwidth was used. While this is reasonable in a lightly-loaded network, it may lead to poor scheduling decisions in a highly-shared network. Prophet has been extended to utilize dynamic resource information provided by a dynamic resource monitor. This information is used to improve processor selection and partitioning for all application types.

The Network Resource Monitoring System (NRMS) has been designed and implemented for the Unix network environment. It runs as a stand alone program that Prophet and other scheduling tools may utilize to improve their scheduling decisions. It consists of a daemon process (DRM) that runs on each computer to monitor and collect system state information and a directory process (DM) that runs on one computer in each processor cluster (Figure 10). The DRM runs continually and writes the state information to a database on disk. The DM keeps an updated table of the resources that reside in the domain of the processor cluster — CPU status, virtual and real memory status, user logins, network latency and bandwidth. Resource attributes such as reliability are also determined for certain resources such as CPUs and network. Unix system calls *vmstat*, *uptime*, *finger*, *sysinfo*, and *ping* are used to provide this information.

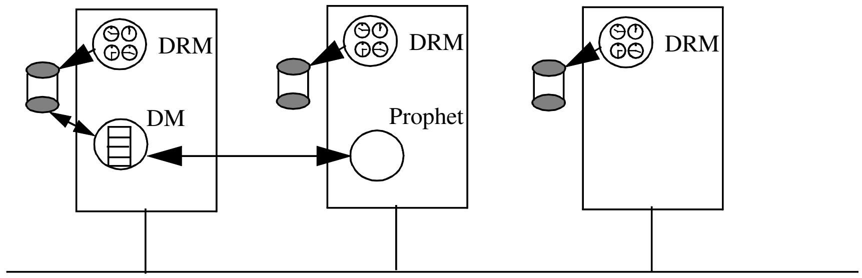


Figure 10: Dynamic resource monitor configuration

Each type of resource is modeled as an object that encapsulates all the information about this type of resource. For instance, a memory object contains available virtual memory space, available real memory space, etc.; a network resource object contains latency and bandwidth, as well as the network type. Some processor clusters have multiple network interfaces such as ATM and Ethernet, and these are treated as distinct network resource objects.

Prophet queries the DM within each processor cluster to get up-to-date information on the state of the cluster resources. NRMS supports a wide-array of query options including information about individual resources such as the CPU load on a particular machine or collective cluster-level information such as the average CPU load for all machines in a processor cluster. A query can also specify a temporal range for information — including the past, present, and future (a prediction). For example, a query might request the average load in a processor cluster right now, the same time yesterday, over the past ten hours, or in the next ten hours. Prediction support will be provided by the Network Weather Service (NWS) and is not yet implemented [18]. NRMS also supports a form of event-based notification. Using this mechanism, a client may indicate that it be informed when the average load in a processor cluster goes above or below a threshold, the network bandwidth increases or decreases beyond a threshold, etc. This type of notification is useful for performing adaptive scheduling. The DM also supports resource location services — e.g., locate an idle Ultrasparc, find user-id “weissman” if logged in, etc.

The NRMS is built on a directory tree which can be dynamically scaled when the system is scaled. It maintains knowledge about all system resources within its monitoring *domain*, where domain is defined in terms of directory, file system, and protection domain. The simplest domain is a set of resources that share a common file system. Prophet uses information provided by the NRMS to provide more accurate scheduling, $\overline{T_{comp}}$ is adjusted to reflect the processor load and $\overline{T_{comm}}$ is adjusted to reflect reduced bandwidth:

$$\overline{T_{comp}} = \frac{comp\_complexity \cdot \overline{arch\_cost} \cdot numPDUs}{p \cdot \overline{cpu\_avail}} \quad \text{(Eq.8)}$$

$$\overline{T_{comm}}[SPMD] = (c_1 \cdot p) + \frac{(c_2 \cdot p \cdot comp\_complexity)}{bw\_avail} \quad \text{(Eq.9)}$$

The term $\overline{cpu\_avail}$ is the average CPU availability as a percentage (idle = 100% available) for all selected processors. Since $\overline{arch\_cost}$ reflects the peak execution time it is scaled by $\overline{cpu\_avail}$ to give the true cost (Eq.8). The number of *PDU*s assigned to each processor will also change as a function of the individual *cpu_avail* values. The term bw_avail is the amount of bandwidth available within the cluster as a percentage (idle network = 100% available). Since the communication parameters reflect the peak communication rates, bw_avail is used to scale the bandwidth-dependent portion of $\overline{T_{comm}}$ (Eq.9). It is also probable that CPU load impacts T_{comm} as reported elsewhere [5], but our model is accurate to 10% on average which gives acceptable scheduling results. The new calculations for $\overline{T_{comp}}$ and $\overline{T_{comm}}$ improve processor selection and data domain decomposition. The equations for the sequential time estimate of the application T_{seq} and the processor weight w_i also change as a function of the CPU load:

$$T_{seq} = \frac{comp\_complexity \cdots numPDUs}{\overline{cpu\_avail}} \qquad \text{(Eq.10)}$$

$$w_i = \frac{1}{(\quad) \cdot \sum_{j=1} \frac{1}{(\quad)}} \qquad \text{(Eq.11)}$$

3.4 Parallel Pipelines

Pipelining is an important source of parallelism in many applications. The archetype example is an image processing pipeline (Figure 11). In this problem, a number of input images are pumped into the pipeline with stages overlapped in execution. The speedup is limited by the pipeline depth and maximum computation time of the pipeline stages. To overcome this performance limitation, the pipeline can be replicated and run in parallel (Figure 12). In the parallel pipeline model (PP) model, each computation stage is a task that will be assigned to a processor. If the pipeline depth is k and there are n pipelines, then kn processors are needed. In the initial version of PP support, each stage must be a sequential computation. In the future, this requirement will be relaxed.

Prophet will determine the appropriate number of pipelines (i.e., processors) to apply to a PP application, and the placement of computation tasks to processors. Unlike the scheduling of SPMD applications, there is no need to partition the data domain. The inputs are served to the ini-

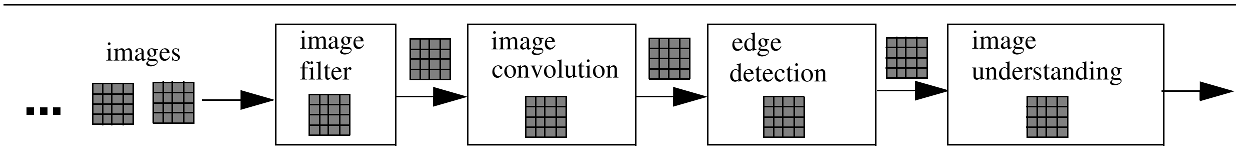


Figure 11: Image processing pipeline

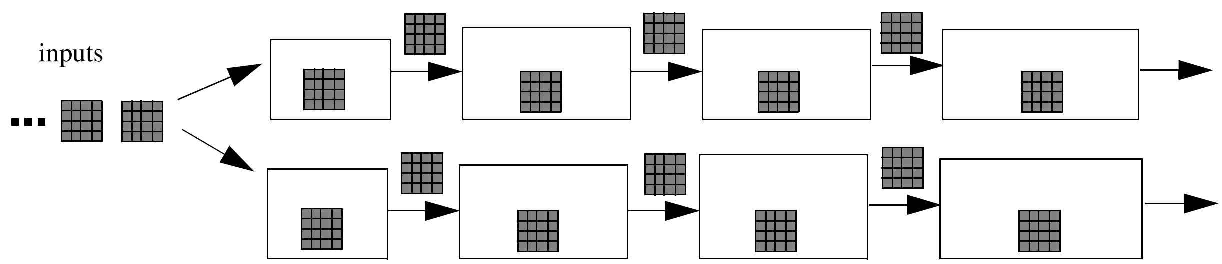


Figure 12: Parallel pipelines

tial stage of the pipelines when they become idle. For PP applications, the *PDU* is the input (e.g., an image), and callback information is provided as before. For *each* computation stage, a *comp_complexity* and *comm_complexity* callback is defined. The number of processors to use depends on the number of inputs (provided by the *numPDUs* callback), the computation granularity of the stages, and the amount of communication contention between the parallel pipelines. Communication cost is more difficult to determine because we cannot assume simultaneous access as in the SPMD case. In PP applications, the stages may differ in the amount of computation and communication and the pipelines are running asynchronously. Consequently, contention is harder to predict. We have observed that the contention is linear in the number of pipelines *np* for short pipelines ($k < 5$)[3], but also depends on the ratio of computation to communication within the pipeline. The greater the ratio, the less likely communications will interfere. A binomial distribution has been used to model the probability of contention with great accuracy.

3. For larger *k*, we speculate that inter-pipeline communication contention might occur. However, most real applications will have a relatively short pipeline depth.

Prophet again explores a series of processor configurations to identify the best schedule. Prophet constructs a cost function for PP application completion time for each candidate schedule:

$$T_{CT}[PP] = num\_inputs \cdot num\_stages \cdot (\overline{T_{comm}} + \overline{T_{comp}}) \qquad \text{(Eq.12)}$$

$$\overline{T_{comp}}[PP] = \frac{(\overline{comp\_complexity} \cdot \overline{arch\_cost})}{kn \cdot \overline{cpu\_avail}} \qquad \text{(Eq.13)}$$

$$\overline{T_{comm}}[PP] = (c_1 \cdot nc) + \frac{(c_2 \cdot nc \cdot comm\_complexity)}{bw\_avail} \qquad \text{(Eq.14)}$$

$\overline{comp\_complexity}$ and $\overline{comm\_complexity}$ are the average *comp_complexity* and *comm_complexity* over the k stages respectively. $\overline{T_{comp}}$ and $\overline{T_{comm}}$ are the average computation and communication time over the k stages respectively. The communication cost is linear in nc, the expected value of the number of simultaneous communications. It has a value on the range from 1 up to the number of pipelines np. It is based on the probability g that a communication in one pipeline will be issued simultaneously with a communication in another pipeline. This probability is estimated to be the percentage of communication relative to computation for a stage on average. It is easily computed from the callbacks. A binomial distribution is used to compute the expected value for the number of simultaneous communications:

$$nc = 1 + \sum_{i=0}^{np-1} i \cdot g^i (1-g)^{np-1-i} \cdot \binom{np}{i} \qquad \text{(Eq.15)}$$

$$g = \frac{\overline{T_{comm}}[PP]}{\overline{T_{comm}}[PP] + \overline{T_{comp}}[PP]} \text{, with } k, n, nc = 1 \qquad \text{(Eq.16)}$$

For example, if the stages performed only communication, then g would be 1.0 and the expected value for the number of simultaneous communications would be np. A binomial distribution is used because there is a constant probability of simultaneity (g) for success/failure, and the pipelines issue messages asynchronously to approximate independence. This communication model has been validated experimentally. Both $\overline{T_{comp}}$ and $\overline{T_{comm}}$ functions are based on full pipelines, i.e., the number of inputs are sufficiently large to fill the pipelines. Since PP communications are rings, task placement is straightforward (see Figure 6). Tasks within the same pipeline will be placed in the same cluster.

4.0 Results

Prophet has been applied to prototype SPMD and PP applications in a heterogeneous LAN environment. We present results for one application in each class. The SPMD application is the five-point stencil used to compute poissons equation on a grid of different sizes. The Mentat implementation of the application was illustrated in Figure 1 with callbacks shown in Figure 4(b). The PP application is a synthetic three stage image processing application with different filter computations specified for the stages. A *m*x*m* convolution mask is applied to each input image as it moves through the pipeline where each pixel is 4 bytes. A mask may be applied iteratively such as to perform thinning or pruning operations [9]. The callbacks are specified in Figure 13 — *N* is the image dimension, m_i is the mask dimension (3, 9, etc.) for stage *i*, $iters_i$ is the number of times m_i is applied, and *num_inputs* is the number of input images. These parameters are all specified as command-line arguments. In this particular PP application, each stage communicates the entire

topology ⇒ ring
comm_complexity ⇒ $4N^2$, $4N^2$, $4N^2$
numPDUs ⇒ num_inputs
comp_complexity ⇒ $iters_1 * m_1^2 * N^2$
⇒ $iters_2 * m_2^2 * N^2$
⇒ $iters_3 * m_3^2 * N^2$

Figure 13: Callbacks for PP application

input image to the subsequent stage in the pipeline. Other PP applications may communicate a reduced or compressed input.

The execution environment consists of three heterogeneous processor clusters on separate sub-networks containing 22 processors. C_1 is an ethernet cluster containing 9 Sparc5s, C_2 is an ethernet cluster containing 9 Sparc2s, and C_3 is an ATM network containing 4 Ultrasparcs. All of these processor clusters are shared with other users. Communication is via Mentat's reliable UDP-based communication library. A range of problem instances has been scheduled using Prophet, see Table 1.

The experimental results for SPMD and PP problems are shown in Table 2. The Prophet schedule contains the selected cluster and the number of processors chosen in that cluster. For PP problem instances, the number must be a multiple of 3 since the application uses 3 stage pipelines. For example, 6 processors would be 2 pipelines. The schedule and completion time of the application is shown when Prophet disables/enables the use of dynamic resource information. The predicted completion time is the time that Prophet estimates the application will require using the selected schedule — it is shown for the case when information use is enabled. The overhead is the time that Prophet takes to compute the schedule. For these problem instances, Prophet opted to use processors in a single processor cluster. For much larger problems under low degrees of sharing, Prophet will use multiple processor clusters if necessary.

Prophet overhead is a small fraction of the overall completion time for all problem instances. The overhead is higher for PP since PP uses a more complex cost evaluation procedure. Problem granularity is larger for higher-numbered problem instances, and Prophet selects a larger number of processors for these problems. For much smaller problems, Prophet explores fewer processor configurations and overhead is often in the range of a 1 millisecond. Prophet also very accurately predicts the application completion time using information about the run-time state of system resources.

Problem instance	Problem parameters	Description
PP-1	N=50, $m_{1..3} = 3$, $iters_{1..3} = 3$	100 50x50 images, each stage performs a 3x3 convolution applied 3 times per stage
PP-2	N=100, $m_{1..3} = 3$, $iters_{1..3} = 5$	100 100x100 images, each stage performs a 3x3 convolution applied 5 times per stage
PP-3	N=100, $m_{1..3} = 3$, $iters_{1..3} = 10$	100 100x100 images, each stage performs a 3x3 convolution applied 10 times per stage
STEN-1	N=200, iters = 100	200x200 input grid, run for 100 iterations
STEN-2	N=400, iters = 100	400x400 input grid, run for 100 iterations
STEN-3	N=800, iters = 100	800x800 input grid, run for 100 iterations

Table 1: Experimental problem instances

In addition, Prophet made better scheduling decisions when it used run-time information (+ info column). The speedups are modest since these applications have a good deal of communication (especially PP), the Prophet data was collected when the system was loaded, and the sequential time reflects the execution time on the fastest single computer when it was idle. It is included as a reference point. The true speedup (i.e., using the same loaded machine for the sequential run) would be higher.

Problem instance	Sequential time	Prophet schedule		Completion time		Predicted completion time	Prophet overhead
		- info	+ info	- info	+ info	+ info	
PP-1	14175	3 in C_3	6 in C_1	15204	7262	7423	14
PP-2	94500	9 in C_1	1 in C_3	44749	34776	36216	14
PP-3	189000	9 in C_1	9 in C_2	125729	93036	94603	14
STEN-1	3000	4 in C_3	4 in C_3	1664	1228	1441	5
STEN-2	12000	4 in C_3	5 in C_1	4670	4260	4079	5
STEN-3	48000	9 in C_1	8 in C_1	19858	13207	13008	5

Table 2: Experimental results. All times shown are in milliseconds. The sequential time and Prophet overhead shown are for the fastest machine type, an Ultrasparc within C1. This was the machine that the application was submitted to. Prophet runs on the submitted machine by default. The sequential time was estimated from the *arch_cost* and *comp_complexity* callbacks and is based on an idle machine.

The experiments were run under a variety of load conditions to explore the benefit of using CPU and network load information. Typically applications[4] that consume both bandwidth and CPU cycles are run in the background to create resource sharing. For each problem instance, Prophet is run with information use enabled (+ info) or disabled (- info -) to compare. When information use is not enabled, Prophet chooses the “best” schedule using static information reflecting the peak CPU and network information. When information use is enabled, Prophet can respond to CPU and network sharing to produce better schedules.

For PP-1, the best schedule is to use 3 ATM-connected Ultrasparcs forming one pipeline in C_3, when it is unloaded. However, a background application was first run in C_3. PP-1 was then scheduled by Prophet with and without information use. With information use enabled, Prophet opted to avoid C_3 due to the observed reduced bandwidth and computation power available, and instead chose to use 6 Sparc 5s in C_1. With information use disabled, Prophet chose the 3 Ultrasparcs in C_3. For PP-2, the granularity is higher and the best schedule is to exploit the additional parallelism available in C_1 (when it is unloaded) since it has more processors available than does C_3. However, a background application was run in C_1 and this caused Prophet to correctly bypass C_1 in favor of C_3 when scheduling PP-2. Prophet does not choose C_2 because the CPU power of the Sparc 2s is too low. Finally, for PP-3, the high granularity again suggests that C_1 be chosen when it is unloaded. However, a background application was run in both C_1 and C_3. In this case, Prophet avoided both C_1 and C_3, and C_2 was selected.

For STEN-1, the best schedule is to use 4 ATM-connected Ultrasparcs in C_3, when it is unloaded. However, a background application was first run in C_3. In this case, the use of information did not affect Prophet’s processor selection (4 Ultrasparcs in C_3 were still selected, probably

4. These background applications are actually instances of our test applications.

because the problem granularity is fairly low), but it did impact the data domain decomposition to the selected processors. A data distribution reflecting the different processor loads prevented a load imbalance. For STEN-2, the best schedule is to again use 4 Ultrasparcs in C_3. However, a background application was run in C_3 and Prophet opted to use 5 Sparc5s in C_1 instead. Finally for STEN-3, the granularity is very high and the best schedule is to use 9 Sparc5s in C_1. However, a background application was run in C_1 and Prophet opted to stay within C_1 but to contract the schedule and use 8 Sparc5s instead. In this case, Prophet also used load information to provide a load balanced decomposition of the data domain.

Clearly for single parallel applications, Prophet produced better completion times when it used dynamic resource information such as processor load and available network bandwidth. This benefit ranged from 10-50% for the problem instances. A related benefit is that Prophet can also produce higher system throughput for sets of jobs. In many cases, Prophet will avoid clusters that are already running CPU- and network-intensive applications. This prevents the newly scheduled job from interfering with currently running jobs, providing a performance benefit to each. Preliminary results for wide-area scheduling of parallel applications also looks promising [17].

5.0 Conclusion and Future Work

Prophet represents a point on the spectrum of scheduling approaches between fully-automated system schedulers and hand-tuned programmer-generated schedules. Prophet demonstrates that it is possible to effectively schedule parallel applications at run-time in shared, heterogeneous LANs and WANs, with limited programmer effort. Prophet exploits the structure of the application and information about the system resources to produce reduced completion time for SPMD and PP application types. The performance of Prophet is superior to system schedulers such as the Mentat scheduler [15], but a direct comparison to hand-tuned programmer-scheduled applications is needed. Our ultimate goal is show performance better than system schedulers and moderately-tuned applications, and within a few percent of highly-tuned programmer-scheduled applications. For the latter, Prophet will be compared with Apples [1].

We are also investigating other applications types to discover the range of applications that can be scheduled automatically. Other parallel application types include bag-of-task (BoT), and combinations of BoT, PP, and SPMD applications. For example, the stages in the pipeline of Figure 12 could be SPMD computations themselves. The suitability of Prophet for more loosely-coupled distributed applications such as metacomputing problems is an open question that we also hope to answer.

Lastly, we are planning to support application adaptivity and re-scheduling using Prophet. For example, if the system resources allocated to an application change dramatically (positively or negatively), the application may need to respond to these changes. We will be investigating how to provide run-time support for adaptivity to user applications. Mechanisms for performing re-scheduling or dynamic load balancing are well known. However, the most effective policies for making re-scheduling decisions have received little attention. Cost models that predict the overhead and benefit of re-scheduling will be developed to support this process.

6.0 References

[1] F. Berman and R. Wolski, "Scheduling From the Perspective of the Application," *Proceedings of the Fifth International Symposium on High Performance Distributed Computing*, August 1996.

[2] T.L. Casavant, and J.G. Kuhl, "A Taxonomy of Scheduling in General-Purpose Distributed Computing Systems," *IEEE Transactions on Software Engineering*, Vol. 14, February, 1988.

[3] K.M. Chandy and E.M. Schooler, "Designing Directories in Distributed Systems: A Systematic Framework," *Fifth International Symposium on High Performance Distributed Computing*, August 1996.

[4] D.L. Eager, E. D. Lazowska, and J. Zahorjan, "Adaptive Load Sharing in Homogeneous Distributed Systems," *IEEE Transactions on Software Engineering*, Vol. 12, May 1986.

[5] S.M. Figueira and F. Berman, "Modeling the Effects of Contention on the Performance of Heterogeneous Applications,"*Proceedings of the Fifth IEEE International Symposium on High Performance Distributed Computing*, August 1996.

[6] I. Foster and C. Kesselman, "Globus: A Metacomputing Infrastructure Toolkit," *International Journal of Supercomputing Applications* (to appear).

[7] A.S. Grimshaw, "Easy to Use Object-Oriented Parallel Programming with Mentat," *IEEE Computer*, May 1993.

[8] A.S. Grimshaw and W. A. Wulf, "The Legion Vision of a Worldwide Virtual Computer," Communications of the ACM, Vol. 40(1), 1997.

[9] A.K. Jain, *Fundamentals of Digital Image Processing*, Prentice-Hall Publisher, 1989.

[10] N. Nedeljkovic, and M.J. Quinn, "Data-Parallel Programming on a Network of Heterogeneous Workstations," *Proceedings of the First IEEE International Symposium on High-Performance Distributed Computing*, Sept. 1992.

[11] H. Topcuoglu et al, "The Software Architecture of a Virtual Distributed Computing Environment," *Proceedings of the Sixth IEEE International Symposium on High-Performance Distributed Computing*, August. 1997.

[12] J.B. Weissman and A.S. Grimshaw, "A Federated Model for Scheduling in Wide-Area Systems," *Proceedings of the Fifth IEEE International Symposium on High Performance*

Distributed Computing, August 1996.

[13] J.B. Weissman, “The Interference Paradigm for Network Job Scheduling,” *Heterogeneous Computing Workshop, 10th International Parallel Processing Symposium IPPS*, April 1996.

[14] J.B. Weissman and A.S. Grimshaw, “A Framework for Partitioning Parallel Computations in Heterogeneous Environments,” *Concurrency: Practice and Experience*, Vol. 7(5), August 1995.

[15] J.B. Weissman, “Scheduling Parallel Computations in a Heterogeneous Environment,” Ph.D. dissertation, University of Virginia, August 1995.

[16] J.B. Weissman and A.S. Grimshaw, “Network Partitioning of Data Parallel Computations,” *Proceedings of the Third IEEE International Symposium on High Performance Distributed Computing*, August 1994.

[17] J.B. Weissman, “Gallop: The Benefits of Wide-Area Computing for Parallel Processing,” UTSA Technical Report, CS-97-8, September 1997.

[18] R. Wolski, “Forecasting Network Performance to Support Dynamic Scheduling,” *Proceedings of the Sixth IEEE International Symposium on High Performance Distributed Computing*, 1997.