Run-time Support for Scheduling Parallel Applications in Heterogeneous NOWs

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

This paper describes the current state of Prophet — a system that provides run-time scheduling support for parallel applications in heterogeneous workstation networks. Prior work on Prophet [10][12] *demonstrated that scheduling SPMD applications could be effectively automated with excellent performance. Enhancements have been made to Prophet to broaden its use to other application types including parallel pipelines, and to make more effective use of dynamic system state information to further improve performance. The results indicate that both SPMD and parallel pipeline applications can be scheduled to produce reduced completion time by exploiting the application structure and run-time information.*[1]

1.0 Introduction

Providing run-time support is crucial for the efficient execution of parallel applications in shared heterogeneous workstation networks. In this environment, scheduling decisions must be deferred until run-time when the status of CPU, network, and memory resources are known. A run-time scheduling system called Prophet has been developed to support the execution of SPMD applications in heterogeneous workstation networks [10][12]. Prophet 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.

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

Prophet represents a departure from traditional approaches to run-time scheduling based on the “simple is best” principle. The many variations of adaptive load sharing algorithms are examples of simple scheduling approaches [2][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. The Prophet approach is to exploit program structure similar to compiler-directed scheduling techniques. Prophet has been integrated into Mentat [5], 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 [11].

Prophet had a number of shortcomings that have now been addressed. Prophet was limited to SPMD applications in local-area ethernet networks, and now supports ATM and wide-area networks [8], and other application types including parallel pipelines (PP). The system was not *overhead-aware* and now provides better control for run-time scheduling overhead. Prophet also assumed limited sharing of CPUs and networks and now supports highly-shared networks. This paper describes scheduling support for PP applications, support for highly-shared networks including sophisticated resource monitoring capabilities, and application-sensitive overhead control. 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 network 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 [7]. 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 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

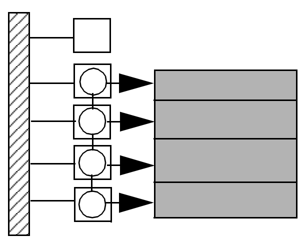

Figure 1: SPMD scheduling. The application is decomposed into a set of data regions (shaded rectangles) each assigned to a task (circle) which is placed on a processor (square). In this example, the tasks are arranged in a 1-D communication topology.

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. Another factor in processor selection is 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 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*.

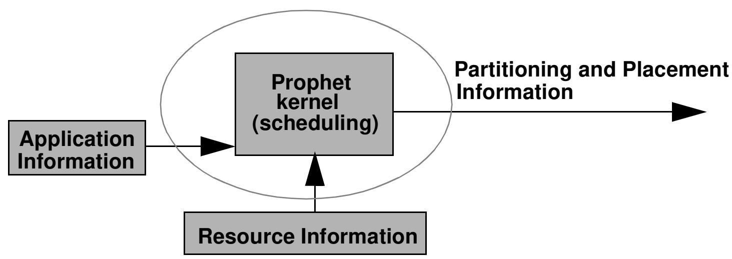

Figure 2: Prophet system architecture

Processor clusters contain homogeneous[2] computers on separate subnetworks (Figure 3). Prophet uses the following cluster information to help guide its scheduling decisions: cluster latency and bandwidth, topology-specific communication cost functions, number of processors, and aggregate processing power. The communication parameters assume peak network performance.

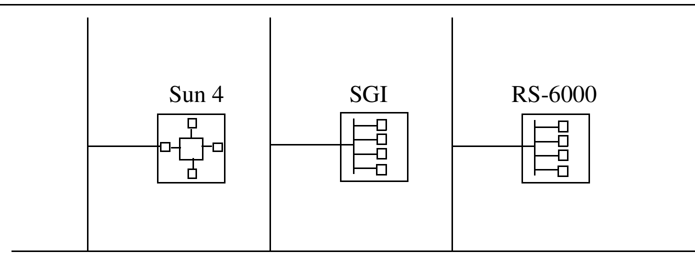

Figure 3: Network organization. A environment containing 3 processor clusters, two connected via ethernet and the other via ATM.

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). The *PDU* (primitive data unit) is the smallest piece of the data domain. For example, it might be a row of a grid as in the stencil application of Figure 1. Callbacks for the stencil application are given in Figure 4(b). The *arch_cost* includes the CPU and

2. Homogeneity is not a strict requirement.

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.

memory overhead for the *PDU* and is based on an idle CPU. A full discussion of callbacks can be found in [10]. A number of *PDU*s will be assigned 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 [10]. Prophet first orders the processor clusters based upon their computation and communication capabilities for the application in order to constrain this search process. The cluster ordering process estimates the computation and communication rates using the cost functions given below. Prophet constructs a cost function for application completion time T_{CT} for each candidate schedule[3]:

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

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

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

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 proces-

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

sors, p, and message size, given by *comm_complexity*. When multiple processor clusters are used, the communication cost functions are combined appropriately [10]. The combined cost function reflects a specific placement of tasks to processors. The placement process exploits information about the application topology and the network topology and is described more fully in [10].

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. This model for T_{CT}, $\overline{T_{comp}}$, and $\overline{T_{comm}}$ has been validated for real SPMD applications in a heterogeneous network environment.

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 a single processor configuration was benchmarked off-line for each architecture type and made available to the scheduling algorithms. The amount of Prophet overhead experienced by the application will be constrained to be proportional to the expected run-time of the application. Prophet overhead will be controlled by limiting the number of processor configurations explored during scheduling, and will depend on the estimated sequential run-time of the application. The estimated sequential run-time of the application can be easily computed from the callbacks. Currently, Prophet overhead must fall within 10% of the estimated sequential run-time of the application.

3.2 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-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 dynamic resource monitor 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 5). 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. Unix system calls *vmstat*, *uptime*, *finger*, and *ping* are used to provide this information. All resources are modeled as objects within the directory. Some

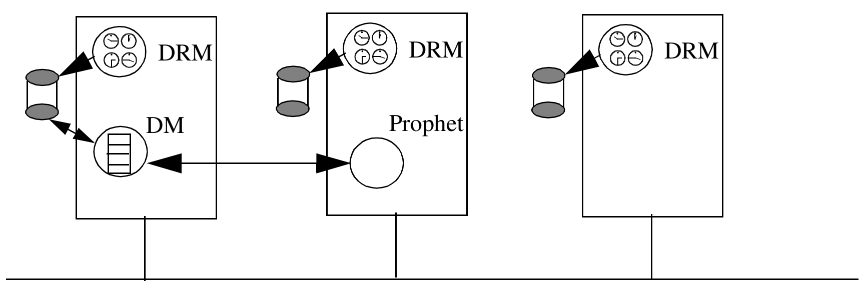

Figure 5: Dynamic resource monitor configuration

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. The DM 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. The DM 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 directory structure is hierarchical and designed to scale to a wide-area system, similar to [3]. The DRM and DM are described more fully in [13].

Prophet uses information provided by the DM 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:

$$\overline{T_{comp}} = \frac{comp\_complexity \cdot \overline{arch\_cost} \cdot numPDUs}{p \cdot \overline{cpu\_avail}} \quad \text{(Eq.4)}$$

$$\overline{T_{comm}}[SPMD] = (c_1 \cdot p) + \frac{(c_2 \cdot p \cdot comp\_complexity)}{bw\_avail} \quad \text{(Eq.5)}$$

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.4). 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.5). The new calculations for $\overline{T_{comp}}$ and $\overline{T_{comm}}$ improve the cluster ordering, processor selection, and data domain decomposition processes.

3.3 Parallel Pipelines

Pipelining is an important source of parallelism in many applications. The archetype example is an image processing pipeline (Figure 6). 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,

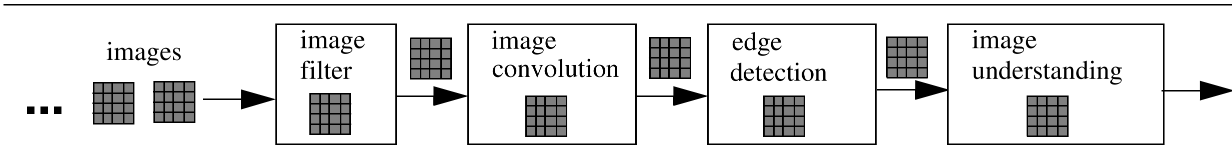

Figure 6: Image processing pipeline

the pipeline can be replicated and run in parallel (Figure 7). In the 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.

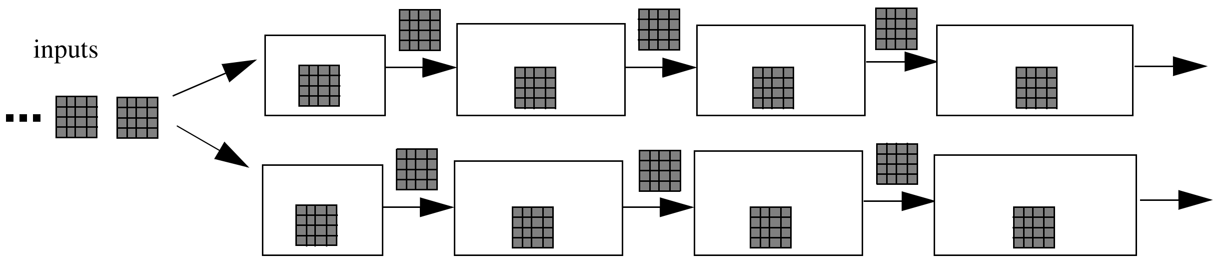

Figure 7: Parallel pipelines

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 initial 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 ($k < 5$)[4], 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.

4. 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:

$$T_{CT}[PP] = num\_inputs \cdot num\_stages \cdot (\overline{T_{comm}} + \overline{T_{comp}}) \quad \text{(Eq.6)}$$

$$\overline{T_{comp}} = \frac{(\overline{comp\_complexity} \cdot \overline{arch\_cost})}{kn \cdot \overline{cpu\_avail}} \quad \text{(Eq.7)}$$

$$\overline{T_{comm}}[PP] = (c_1 \cdot nc) + \frac{(c_2 \cdot nc \cdot comm\_complexity)}{bw\_avail} \quad \text{(Eq.8)}$$

$\overline{comp\_complexity}$ 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:

$$nc = 1 + \sum_{i=0}^{np-1} i \cdot g^{i}(1-g)^{np-1-i} \cdot \binom{np}{i} \quad \text{(Eq.9)}$$

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*. 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.

4.0 Results

Prophet has been applied to prototype SPMD and PP applications in a heterogeneous network environment. 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. Each pixel is 4 bytes. A mask may be applied iteratively such as to perform thinning or pruning operations [6]. The callbacks are specified in Figure 8 — *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

topology ⇒ ring
comm_complexity ⇒ $4N^2$, $4N^2$, $4N^2$
numPDUs ⇒ num_inputs
comp_complexity ⇒ $iters_1 * m_1^2 * N^2$
⇒ $iters_2 * m_2^2 * N^2$
⇒ $iters_3 * m_3^2 * N^2$
arch_cost ⇒ Sparc5: = .0008
⇒ Ultrasparc: = .0007
⇒ Sparc 2: = .003

Figure 8: Callbacks for PP application

parameters are all specified as command-line arguments. In this particular PP application, each stage communicates the entire input 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. All 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.

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=200, iters = 100	800x800 input grid, run for 100 iterations

Table 1: Experimental problem instances

Prophet overhead is a small fraction of the overall completion time for all problem instances — it is higher for PP since PP uses a more complex cost evaluation procedure. Since problem granularity is larger for higher-numbered problem instances, a larger number of processors will be selected by Prophet. 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. 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.

The experiments were run under a variety of load conditions to explore the benefit of using CPU and network load information. Typically applications[5] 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.

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

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.

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 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. 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.

5.0 Conclusion and Future Work

Prophet demonstrates that it is possible to effectively schedule parallel applications at runtime in shared, heterogeneous, workstation networks, 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. We are continuing to build upon Prophet to expand its functionality. A prototype for scheduling parallel applications using Prophet in a wide-area environment has been implemented [8], and experimentation is beginning.

We are also exploring techniques to make Prophet more extendable to other application types by redesigning the library software to be more “object-oriented”. Additional application types that will be supported in the near future include bag-of-task (BoT), and combinations of BoT, PP, and SPMD applications. For example, the stages in the pipeline of Figure 7 could be SPMD computations themselves. It is also common for scientific applications to have multiple coupled SPMD computations. We are planning to handle these types of applications in the future. An object-oriented structure would allow a combined scheduling model to be easily formed as a combination of existing models, or would allow a new scheduling model to derived from others. For example, a SPMD scheduling model for irregular applications could be derived from the default SPMD scheduling model. We are also planning to provide greater support for scheduling multiple applications that may compete for system resources. This work will be based on an interference model that has been developed previously [9]. 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.

6.0 References

[1] F. Berman and R. Wolski, "Scheduling From the Perspective of the Application," *Fifth International Symposium on High Performance Distributed Computing* (*HPDC-5*), 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* (*HPDC-5*), 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] A.S. Grimshaw, "Easy to Use Object-Oriented Parallel Programming with Mentat," *IEEE Computer*, May 1993.

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

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

[8] J.B. Weissman and A.S. Grimshaw, "A Federated Model for Scheduling in Wide-Area Systems," *Fifth International Symposium on High Performance Distributed Computing* (*HPDC-5*), August 1996.

[9] J.B. Weissman, "The Interference Paradigm for Network Job Scheduling," *Heterogeneous Computing Workshop, 10th International Parallel Processing Symposium IPPS*, pp. 38-45, April 1996.

[10] J.B. Weissman and A.S. Grimshaw, "A Framework for Partitioning Parallel Computations in Heterogeneous Environments," *Concurrency: Practice and Experience*, Vol. 7(5), pp. 455-478, August 1995.

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

[12] J.B. Weissman and A.S. Grimshaw, "Network Partitioning of Data Parallel Computations," *Proceedings of the Third International IEEE Symposium on High Performance Distributed Computing*, pp. 149-156, August 1994.

[13] X. Zhao and J.B. Weissman, "The Design of a Dynamic Resource Monitor," UTSA Technical Report, CS-97-2, January 1997.