Prophet: Automated Scheduling of SPMD Programs in Workstation Networks

Jon B. Weissman
Division of Computer Science
University of Texas at San Antonio
San Antonio, TX 78249
USA
(210-458-5689 voice, 210-458-4437 fax)
(jon@cs.utsa.edu)

Abstract

Obtaining efficient execution of parallel programs in workstation networks is a difficult problem for the user. Unlike dedicated parallel computer resources, network resources are shared, heterogeneous, vary in availability, and offer communication performance that is still an order of magnitude slower than parallel computer interconnection networks. Prophet, a system that automatically schedules data parallel SPMD programs in workstation networks for the user has been developed. Prophet uses application and resource information to select the appropriate type and number of workstations, divide the application into component tasks and data across these workstations, and assign tasks to workstations. This system has been integrated into the Mentat parallel processing system developed at the University of Virginia. A suite of scientific Mentat applications have been scheduled using Prophet on a heterogeneous workstation network. The results are promising and demonstrate that scheduling SPMD applications can be automated with good performance[1].

Keywords: *scheduling*, *network computing, data parallel*

1. This work has been partially funded by grants NASA NGT-50970 and NSF ASC-9625000.

1.0 Introduction

Parallel processing in heterogeneous workstation networks has become a viable alternative to the use of traditional parallel machines for many applications. Advances in network-based toolkit technologies including PVM [10], Charm [9], and Mentat [6] and advances in network technologies including ATM and Myrinet, have made the NOW (network of workstation) environment accessible and attractive for parallel computing. However, the NOW environment presents a number of unique challenges to obtaining good performance. The workstation network is shared, contains heterogeneous computers that may vary in availability, and uses interconnection networks that still lag well behind the performance of parallel computer interconnection networks.

Resource sharing impacts the way in which the parallel computation can best be decomposed across the selected workstations. Workstations should be given a portion of the computation that is proportional to their computational power, since the computational load must be balanced across the selected workstations. If workstations are heterogeneous, then a highly loaded fast workstation may be preferable to an idle, but weaker workstation. Another dimension of heterogeneity is the network. Contemporary NOWs may contain several kinds of networks including backplane-connected clusters, slow- and fast-ethernet, FDDI, ATM, Myrinet, etc.

In the NOW environment it is often necessary to constrain parallelism to reduce the high cost of communication. The appropriate number of selected workstations though depends on the application characteristics, network performance, and workstation characteristics. It is possible to automate the complex process of workstation selection and problem decomposition for an important class of parallel applications known as SPMD (single-program-multiple-data), see Section 3.0. It has been shown that parallel applications implemented in an SPMD-style can achieve efficient execution in a NOW environment [7]. The principle reason is that it is relatively easy to control the granularity of such applications and match it appropriately to the network environment. The SPMD scheduling process consists of workstation selection, task placement, and data partitioning. Each of these factors contribute to reduced completion time for the application, but are difficult for the programmer to do well.

Effective scheduling requires information about the application and network resources. Information about the network resources is provided by a combination of off-line benchmarks, run-time probes, and static configuration data. Obtaining application information is a more challenging problem that requires the participation of the programmer. A simple API that allows the programmer to specify the salient application details has been developed. This API is applicable to a large class of SPMD applications and is fairly easy to use.

A system called Prophet that automates the scheduling process for SPMD applications to obtain reduced completion time has been developed [12]. Prophet uses network resource information and application information to guide the scheduling process, and has been integrated into Mentat, an object-oriented parallel processing system developed at the University of Virginia [6]. Prophet is unique in that it addresses the problems of workstation selection, partitioning, and placement together. Workstation selection is performed to exploit workstation heterogeneity and to achieve an appropriate computation granularity; placement is performed to limit the amount of communication overhead, and partitioning is performed to achieve load balance. Other systems address subsets of these problems, see Section 2.0. An earlier paper established the efficacy of the Prophet scheduling heuristics in simulation [11]. This paper describes the Prophet system architecture and presents experimental results for a suite of four scientific applications.

2.0 Related Work

A number of research groups are addressing run-time scheduling of SPMD applications in workstation networks. These projects can be divided into static [1][2][6][9] and dynamic

[3][4][5][8] approaches. Static schemes make a runtime scheduling decision, but do not re-evaluate or re-schedule the application as it is running. Dynamic schemes will re-schedule the application in response to runtime factors including shifts in processor or network load, machines becoming available or unavailable. Static approaches spend more time trying to pick the best schedule since the decision is fixed, while dynamic approaches are often simpler they are employed more frequently and thus generate higher overhead.

Mentat [6] and Charm [9] both use a variant of load sharing in which a subset of machines are probed to determine the machine with the lightest load. Each task is scheduled individually in these models. Co-scheduling approaches as in [1] consider a SPMD application as a collection of tasks that must be scheduled together. Issus of partitioning are left to the programmer in these systems. Apples [2] is a scheduling framework in which application and network resource information is used by a programmer provided scheduling agent. This agent may perform partitioning and placement for SPMD applications. In some sense, Prophet is most similar to Apples, but offers a higher degree of automation since a scheduling agent is not required.

Adaptive parallelism [3] supports a bag-of-task model in which processors execute tasks from a shared task set. In this model, processors may be allocated and de-allocated dynamically. The scalability of this paradigm for large numbers of processors and tightly-coupled SPMD applications appears to be a problem. DataParallel C [8] and DAME [5] support dynamic load balancing based on CPU load for data parallel applications. In these systems, load balance is achieved by redistributing data among the processors. DPC adopts a compiler-directed approach, while DAME provides a library of load balance routines. DPC is limited to regular data parallel applications. PVM has been extended to support several kinds of adaptive load distribution for parallel applications [4]. One method, ADM, is similar to DPC and DAME, while MPVM supports transparent migration of process-based VPs. These dynamic scheduling methods may outperform static methods in the presence of high CPU load variability for long-running applications. However, these methods do not address the workstation selection and task placement problems, and generally do not consider network communication costs.

3.0 Coarse-Grain Data Parallelism

Data parallelism is a paradigm for expressing the natural parallelism inherent in problem domains. In data parallel applications, the same basic operation may be applied to all elements of the data domain in parallel. Many science applications are naturally data parallel and the archetype data parallel application is a fine-grain grid-based problem such as arises from the discretization of continuous partial differential equations (PDE's). When data parallel applications are implemented on distributed-memory multicomputers or workstation networks they are typically written using a coarse-grain data parallel paradigm known as single-program-multiple-data or SPMD [7]. SPMD relaxes the lock-step synchrony of fine-grain data parallelism. The basic idea is that it is not profitable to try to exploit all available parallelism in the problem (i.e., execute all data elements in parallel). Instead, a set of identical tasks will be created with each assigned a chunk of the data domain. Two examples of coarse-grain SPMD problems are matrix-vector multiplication and gene sequence comparison, see Figure 1.

In the SPMD model, a single task will be placed on a processor and tasks are loosely-synchronous. The tasks will execute in parallel, but will operate on their portion of the data domain sequentially. It is more efficient since a communication is not needed for each and every data element, but rather the task can do a single communication for all of its contained data elements required by its neighbors to maintain data-dependence relationships. This results in fewer, larger communications that have better performance. For grid-based problems in particular, it is not difficult to transform a fine-grain problem into SPMD as shown in Figure 2. Here, each task is assigned some number of rows of the grid and the tasks communicate by exchanging entire rows that border each other.

4.0 Prophet

Prophet implements a three-stage process to automate the entire scheduling process for data parallel SPMD applications, *resource availability*, *scheduling*, and *instantiation*, see Figure 3. Resource availability determines the run-time state and status of all workstations in the network of the network. It is accomplished by run-time daemons that monitor the state of the network. Scheduling ascertains the set of selected workstations based on information from the resource availability stage, a partitioning of the data domain across these workstations, and the placement of tasks to the workstations. It is accomplished by the Prophet kernel which is a library linked with the user's application.

Workstation selection, partitioning, and placement information is returned in a set of data structures. Instantiation initiates the run-time execution of the application using this information and is accomplished by language- and application-dependent mechanisms. The Mentat version of the instantiation process is described shortly. The Prophet kernel requires two additional sources of information to schedule the application, see Figure 3. Information about the communication and computational structure of the application and *static* information about the network resources must be provided. The network information includes the network configuration, workstation speeds, communication cost functions, router and format conversion cost functions.

Application Information

Information about the data parallel application is central to the scheduling process. Within the application, the data domain is decomposed into a number of primitive data units or *PDU*s, where the *PDU* is the smallest unit of data decomposition. The *PDU* is problem and application specific. For example, the *PDU* might be a row, column, or block of a matrix in a matrix-based application.

Data parallel applications typically execute a series of alternating computation and communication phases. Most data parallel computations are iterative with the computation and communication phases repeating after some number of phases. This is known as a *cycle*. Information about each computation and communication phase is provided by *callback* functions. The callbacks are a set of run-time functions that provide critical information about the communication and computational structure of the application and are invoked during the scheduling process. Each computation phase must have the following callbacks defined:

- *numPDUs*
- *comp_complexity*
- *arch_cost*

The number of *PDU*s manipulated during a computation phase, *numPDUs*, depends on problem parameters (e.g., problem size). The amount of computation (in number of instructions) performed on a *PDU* in a single cycle is known as the computation complexity, c*omp_complexity*. The architecture-specific execution costs associated with *comp_complexity* are captured by *arch_cost,* provided in units of µsec/instruction. The *arch_cost* contains an entry for each workstation type in the network. To obtain the *arch_cost*, the sequential code must be benchmarked on each workstation type. Each communication phase must have the following callbacks defined:

- *topology*
- *comm_complexity*

The topology refers to the communication topology of the application, e.g., *1-D*, *tree*, etc. The amount of communication between tasks is known as the communication complexity, *comm_complexity*. It is the number of bytes transmitted by a task in a single communication during a single cycle of the communication phase. For irregular applications, the callbacks are typically provided as an average. Among the computation and communication phases, two phases are distinguished. The *dominant computation* phase has the largest computational complexity and the *dominant communication* phase has the largest communication complexity[2]. An example of the callbacks for the *1-D N*x*N* five-point stencil, a regular data parallel application, is shown in Figure 4. This application was implemented according to the picture in Figure 2.

Resource Information

Information about the network resources is needed to make the best possible scheduling decisions. The heterogeneous network is organized as a collection of homogeneous *processor clusters*, see Figure 5. A homogeneous processor cluster contains machines of similar peak computational power. This network contains three processor clusters containing Suns, SGIs, and RS-6000s respectively. A processor cluster contains private communication bandwidth with respect to other clusters and may be connected by ethernet (fast or slow), FDDI, ATM, or other network types. Thus, the network resources exhibit heterogeneity in terms of computation and communication capacities. Processor clusters are themselves connected by routers, gateways, or bridges.

Information about the processor cluster is stored by a workstation that is denoted as the *manager* (darkened workstation). The *manager* stores the following information:

- *interconnection LAN*
- *workstations*
- *aggregate power*
- *communication functions*

The *interconnection LAN* of the cluster refers to the LAN type (ethernet, ATM, etc.). The *workstations* include the number and type of the available workstations, and their load status. In the future, network load information will be collected as well. The *aggregate power* is the total

2. The dominant phases may be problem or run-time dependent.

Mflops and Mips available. The *communication functions* predict the communication cost as a function of message size and number of workstations for the cluster. They are of the following form for ethernet LAN: $T_{comm}(b, p) = c_1 + c_2 p + b(c_3 + c_4 p)$ where c_1 and c_2 are latency constants, c_3 and c_4 are bandwidth constants, p is the number of communicating workstations, and b is the message size. A suite of communication programs has been developed to automatically produce these cost constants.

The callbacks are invoked to predict the execution cost for a particular schedule. For example consider the *1-D* stencil of Figure 4. Suppose that N=100, the candidate schedule contains four Sun 4 IPC's, and the *1-D* communication function for the IPC cluster is 1.2 + .46p + b(.0004 + .0005p) msec. The amount of time the application spends in computation given this schedule would be: 25 * 5(100) * .0006 = 7.5 msec, as each workstation is given 25 *PDU*s (rows) and each row requires 500 floating point operations with an *arch_cost* of .0006 for the IPC (see Figure 4). The amount of time the application spends communicating given this schedule would be: 1.2 + .46(4) + 400(.0004 + .0005*4) = 4.0 msec.

Scheduling Process

Scheduling is a heuristic process that is performed to achieve reduced completion time for the SPMD application. It utilizes application and resource information as described above to guide this process. Reduced completion time requires selecting the appropriate type and number of workstations, a load balanced decomposition of the data domain across these workstations, and a task placement that minimizes communication overhead.

Selecting the best workstations is accomplished by an algorithm which orders the processor clusters based on an estimate of how long the application will take using each cluster. What is best depends on the application characteristics. Both computation and communication properties of the application and the clusters are considered. Choosing the best number of workstations within a cluster also depends on the application and cluster characteristics. For example, a very large problem may more effectively utilize a larger number of processors than a smaller one. The algorithm searches the space of workstations to determine the best number. For each set of workstations explored, the algorithm estimates the computation and communication costs both of which can be calculated using the information given above.

The algorithm is based on three cost measures computed from the callbacks: T_{comp}, the average amount of time spent in computation per iteration, T_{comm}, the average amount of time spent in communication per iteration, and T_c, the average amount of time spent per iteration total. The algorithm tries to minimize T_c. If there is no overlap between T_{comp} and T_{comm} then T_c is simply the sum of T_{comp} and T_{comm}. These measures are a function of the number of workstations selected in each cluster. The algorithm explores the clusters linearly from estimated best to worst using the best configuration up to cluster *i-1* as a starting point for cluster *i*. The ordering algorithm uses T_c to first order the clusters and each cluster is explored in two phases. For each cluster explored, the first phase adds workstations from the current cluster in order to reduce the T_{comp} component of T_c (this may cause T_{comm} to increase) and the best T_c value is recorded.

In the second phase, T_{comm} is targeted for reduction. In this phase, workstations are added for the current cluster while removing workstations from the cluster than contributed the largest communication cost. This is guaranteed to reduce T_{comm}, but the impact on T_c is unpredictable since T_{comp} may increase since potentially faster workstations are being traded for slower ones. The cluster that contributes the largest communication cost may change during the course of phase 2 as workstations are traded. The configuration that yields the minimum T_c after both phase 1 and phase 2 is stored. This is the starting configuration that is used as the next cluster is considered.

The algorithm explores all clusters. The worst-case order of this algorithm is O(mP): O($mlog_2P$) for cluster ordering, O($mlog_2P$) for phase 1, and O(mP) for phase 2, for m clusters and P total processors. For each cluster it locates the best number of processors by a form of binary search, the details of which are provided in [12]. In practice, the amount of computation performed for each configuration is very small and the linear dependence on P is mitigated to a certain extent. Future optimizations include setting a limit on the number of configurations explored based on the predicted overhead and run-time length of the application.

Once the workstations are selected, a load balanced decomposition of the data domain is calculated. Each workstation is given a share of the data domain that is proportional to its power, where power may be application and problem size dependent. For example, a workstation with a very fast floating point unit may offer much better performance for a floating-point intensive application than for an integer-dominated application. Furthermore, the workstation performance may be dependent on problem size due to cache effects. All of these factors are taken into account. The decomposition is computed and stored in a structure called the *partition_map* — it contains an entry for each selected workstation (k total) that refers to the number of *PDU*'s to be assigned to each workstation, see (Eq.1). All of the entries in A_i must sum to *NumPDU*s. This requires that fractional entries be rounded to the nearest integer[3]. The power weight w_i is calculated from application callback information and may be problem-size dependent [11]. A smaller w_i reflects a faster machine. The *partition_map* reflects a logical decomposition of the data domain. During instantiation, the *partition_map* is used to physically decompose the data to the tasks.

3. The rounding has a marginal impact on load balance for most applications.

$$A_i = \frac{NumPDUs}{\sum_{j=1}^{k} \frac{w_i}{w_j}} \qquad \text{(Eq.1)}$$

Once the workstations are selected and the data decomposition determined, a task will be assigned to each selected workstation to carry out the computation. In the SPMD model, a single task is placed per workstation. The placement process is performed to reduce potential communication costs. Although the SPMD tasks are identical, they may communicate in a pattern that can be exploited to reduce these costs. One of the primary sources of communication overhead in the network environment is communication that crosses a router or gateway. This will occur for communications between workstations in different clusters. However, using application topology information (as provided by the *topology* callback), tasks may be assigned to workstations in different clusters in a manner that minimizes the number of such communications. For example, consider a *1-D* and a *tree* application communication topology using three processor clusters C_1, C_2 and C_3 in Figure 6. Note that only the shaded tasks communicate across the router — this is the best possible placement assuming the communication capacity on inter-cluster links is smaller than the communication capacity within any cluster, a reasonable assumption for most networks.

The Prophet system architecture is not tied to a particular parallel/distributed software or hardware system. To implement the Prophet architecture, a basic set of system capabilities are required [12]. An implementation of Prophet in the Mentat parallel processing system targeted to the NOW environment is described next.

Mentat Implementation

Mentat is an object-oriented parallel processing system based on C++. Mentat programs are written in an extended C++ language called MPL. The programmer specifies granularity information to the system by indicating that a class is a Mentat class. Instances of Mentat classes, known as Mentat objects, are implemented by address-space disjoint processes, and communicate via methods. Method invocation is accomplished via an RPC-like mechanism.

Data parallelism is expressed in Mentat by defining a Mentat class that corresponds to the SPMD task and instantiating some number of Mentat objects of this class. In Mentat, the programmer is responsible for choosing the number of tasks and decomposing the data domain. Mentat does automate the placement of Mentat objects to processors but does not use any program information to do so. In the remainder of the paper, the term object and task are used synonymously. For more details about Mentat, the reader is referred to [6]. The Mentat implementation of each piece of the system architecture in Figure 3 is now described.

Prophet Kernel

The Mentat implementation of the Prophet kernel is written in C++ and is compiled in with the Mentat run-time system library also written in C++. All applications components link this library. The scheduling algorithm is implemented in C++ within the kernel. The kernel also manages the set of objects created by instantiation. It treats the set of objects as a collection, and defines two useful variables that the task implementation can use, `COLLECTION_ID`, the id of the object, and `NUM_COLLECTION`, the number of objects in the collection. In the Mentat implementation, this id maps into the Mentat object name, which is needed to enable communication between objects.

Resource Information and Configuration

The network model of Figure 5 is easily implemented in Mentat which already defines a notion of cluster. The communication and router cost functions are specified by including a set of latency and bandwidth constants within the Mentat configuration file [12]. These values allow Prophet to construct the appropriate communication cost functions for workstation networks. The per-byte cost of format conversion is specified as well. The communication parameters are determined by benchmarking a set of topology-specific communication programs written using the Mentat communication system. It is possible that some clusters will provide multiple network interfaces such as ATM and ethernet. In this case, multiple communication parameters could be specified. Supporting multiple network interfaces is the subject of future work.

The Mentat system runs a daemon process known as the *instantiation manager* on each host in the configuration that is responsible for collecting load information. One *instantiation manager* per cluster is designated as the *manager* in the model. The current implementation of dynamic resource availability is based on a sender-initiated probe of all hosts in the configuration to obtain their load status[4]. When a scheduling request arrives, load and availability information is determined and an aggregate of this information is made available to Prophet.

Application Information Implementation

A C++ callback API has been implemented by defining an abstract base class `domain`, see Figure 7. The callbacks are member functions on this class. The definitions of the argument types are omitted. The structure PV is a parameter vector and is similar to argv in C. It may contain any number of problem parameter values needed to implement the callbacks. Currently the programmer marshals the problem parameters into PV and instantiates the domain with PV as an argument. An example is provided in the next section. The computation and communication phases are represented as integers and it is up to the implementation of the domain class to map these integers into the program phases.

The domain class must be derived and implemented for a particular data parallel application. For example, a domain class for a stencil application, `stencil_domain`, that provides information about the *1-D* stencil application has been defined. The implementation of the

4. A more scalable load collection strategy is planned.

`comp_complexity()` callback for the stencil application is shown in Figure 8. This application has one computation phase, hence the simple switch statement. This callback depends on a single problem parameter, the problem size *N* that is extracted from the parameter vector PV.

Callback specification requires a small amount of information in most cases. In particular, for regular SPMD applications it is straightforward. This mechanism relies on the fact that the programmer generally knows the code details, algorithms, and application structure, and should be able to provide this information fairly easily. However, for irregular applications, callback specification becomes slightly more complex. Irregular applications may have one or more of the following properties: the amount of computation varies per *PDU*, the total amount of computation varies from cycle to cycle, or the amount of data transmitted varies from cycle to cycle. In these cases, *comp_complexity* or *comm_complexity* must be provided as an average. Several of the applications presented later are irregular. Surprisingly, a simple average measure for these callbacks leads to very accurate scheduling. Applications consisting of multiple coupled data parallel computations can also be handled by provided callbacks for each computation. An example of this is a finite-element application presented later.

The programmer effort involved in developing the callbacks include a small amount of benchmarking to determine *arch_cost* - the insertion of a simple timer suffices for most applications. If the programmer does not know *comp_complexity* or *comm_complexity* off-hand, probes can be inserted into the application to determine instruction counts and message sizes respectively. The programmer effort required to develop the callbacks may be mitigated by two factors. First, this effort may be amortized since the program may be run multiple times using different data sets. Second, the callbacks are portable and expose application information which could promote re-use. For example, the callbacks for Gaussian elimination or a Conjugate gradient solver could be re-used within another application. In fact, a library of callbacks for common applications could be developed and provided to the programmer. Instead of callback specification, the programer could opt to hand-tune their application. If the best possible performance is desired, then this course is recommended. However, it is speculated that the programmer would need an even deeper knowledge of the application to optimize its performance, and a greater degree of benchmarking would be required.

The programmer does have the option of omitting portions of the callbacks to simplify the specification, but may risk scheduling inaccuracies. They must specify *comp_complexity* and *comm_complexity* for the dominant phases, and *numPDUs*. If the dominant phases far out-weigh the other phases then this may work fine. Otherwise, inaccuracies may arise. The programmer could incrementally introduce other phases and see if the resulting performance is adequate. Sophisticated programmers could introduce data-dependent callback information to further improve accuracy. For example, *arch_cost* may be different for different problem-sizes. Prophet can handle applications with data-dependent callbacks, but only if these are statically known. The reason is that the callbacks are called once at the start of application execution. Applications in which *comp_complexity* or *comm_complexity* change dynamically and in an unpredictable manner such as in adaptive mesh problems would be difficult for Prophet to handle well. Similarly, problems with topologies that place constraints on the number of workstations such as 2D problems are currently not supported, though this capability could be added.

Instantiation

Instantiation is the process of initiating the application execution using the scheduling information provided by the Prophet kernel. In the Mentat implementation, the program interface to the Prophet kernel is provided by a function `partition` that returns the partition and placement information in a set of data structures, see Figure 9. A Mentat facility to support instantiation, `DP_create`, that instantiates a Mentat object (i.e., a task) on each selected workstation, and communicates the list of created objects to each object, has been provided. This facility allows the object implementation to establish the communication topology and to determine its communicating partners.

A partial main program for *1-D* stencil written in MPL is presented in Figure 10. The main program begins by constructing PV and creating the `stencil_domain` object. The `stencil_domain` object is then passed to `partition` — this will enable Prophet to invoke the callbacks. A call to `DP_create` is then made to place a Mentat object on each workstation based on the information contained in PR. The definition of `partition_rec` is omitted since it is a fairly complex structure. The objects are instances of the Mentat class `sten_worker`. `DP_create` returns the list of created Mentat objects.

The information returned by `partition` is also needed for data decomposition. In this problem, the *partition_map* is used to physically decompose the grid into 1-D chunks via the call to `1D_carve`. The implementation of stencil relies on facilities in a library that manages 1-D and 2-D data structures known as `DD_array`. The grid is represented as a memory-contiguous 2-D array of floating point values, `DD_floatarray`. The implementation of `1D_carve` uses library facilities to extract the appropriate pieces of the grid.

Alternative Implementations

It would be possible to integrate Prophet into other parallel processing systems such as PVM, and possibly in another language such as Fortran. A port to another system would require that the following software components be re-written: the prophet kernel (about 3000 lines of code), the callback interface (changing from C++ classes to a Fortran or C data-structure), resource information and configuration (run-time data collection processes would need to be implemented), and instantiation (the host language and system must support dynamic process creation and messagepassing of some kind). The Prophet library interface such as shown above in Figure 10 would not need to change: `partition` and `DP_create` could easily be Fortran or PVM facilities.

5.0 Results

Prophet has been applied to a suite of four data parallel applications of differing structure spanning a range of problem sizes. The results show that good performance is obtained while Prophet overheads are tolerable. In particular, automated data domain decomposition and task placement are critical to achieving this performance. The test suite includes: Gaussian elimination with partial pivoting (GE), a canonical five-point stencil code (STEN), a large-scale finite-element code (FEM), and a gene-sequence comparison code (CL). The latter two applications are significant codes that solve real problems in computational physics and biology respectively.

The experimental heterogeneous environment is a local-area ethernet-connected network of workstations. The environment contains three processor clusters: C_1 contains 6 SGI Indigo's (based on the MIPS R4000), C_2 contains 8 Sun Sparcstation 2's, and C_3 contains 8 Sun4 IPC's all joined by a router. There are no endian or data format differences for standard data types between these machines and no explicit conversion is needed. A set of synthetic endian conversion routines have been implemented to explore this overhead. These clusters are shared with other users and experimentation was performed when the clusters were lightly loaded. High degrees of resource sharing could impact the quality of Prophet's scheduling decisions[5]. All data presented is the result of multiple program runs with average values shown.

The Mentat Applications

The five-point stencil code (STEN) is based on an iterative PDE solver that is used to solve Laplace's equation: $u_{xx} + u_{yy} = 0$ on the unit square. Using finite-differences a grid is imposed

5. The new version of Prophet now supports higher degrees of resource sharing.

over this domain with the grid points u_{ij} related in the following way: $-u_{i+1,j} - u_{i-1,j} - u_{i,j+1} - \quad {}_{\cdot,j} \quad + \quad {}_{\cdot,j} = \qquad = 1, \quad , N$. A grid of size N produces a linear system that contains N^2 equations corresponding to N^2 interior grid points. In the parallel implementation of STEN the *PDU* is defined to be a row of the grid, and the tasks are arranged in a *1-D* communication topology. The tasks iteratively execute a single dominant computation phase where the grid points are updated according to the rule given above, followed by a dominant communication phase where the tasks exchange north and south borders of the grid.

The finite-element application (FEM) solves an electromagnetic-scattering (EM) problem for the electromagnetic fields in the vicinity of a set of scatterers. A finite-element mesh is imposed on the problem domain and a 2-D Helmholtz equation is transformed into a system of linear equations: $\mathbf{K} \cdot \mathbf{d} = \mathbf{F}$. FEM contains two dependent data parallel computations that are executed sequentially, *assembly* and *solve*. Prophet is applied to each in turn. In the assembly phase the stiffness matrix $\mathbf{K}$ and the force vector $\mathbf{F}$ are computed with the *PDU* being a single finite-element. The stiffness matrix that results is large, very sparse, and symmetric. The solve computation uses a bi-conjugate gradient solver BCG to solve the linear system with the *PDU* being a row of the stiffness matrix. The stiffness matrix is first preconditioned by diagonal scaling to improve convergence. In the parallel implementation of assembly the finite-element mesh is decomposed across a set of tasks that compute contributions to the stiffness matrix and force vector. Assembly communication is a broadcast topology. In the parallel implementation of BCG the stiffness matrix and a set of vectors are decomposed across another set of tasks. BCG requires both *1-D* and *tree* communication.

Complib (CL) is a biology application that classifies protein sequences that have been determined by DNA cloning and sequencing techniques. CL compares a source library of sequences to a target library of sequences. Comparing protein or DNA sequences is a string matching problem over strings of base pairs (A, C, G, T). CL utilizes three heuristics for string matching, Smith-Waterman, a rigorous dynamic programming algorithm, Fasta, a fast heuristic that improves performance 20-100 times, and Blast, another fast heuristic. A set of DNA sequence libraries randomized for load balance have been used with the Smith-Waterman (SW) and Fasta (FA) algorithms. The details of these algorithms may be found in [12]. In the parallel implementation of CL, the source library is decomposed across a set of tasks with the *PDU* being a single source sequence. Each task compares all of the sequences it is assigned to a sequence in the target library during a single iteration. The tasks are arranged in a tree with the leafs performing the computation which generates a comparison score for the current target based on the source sequences.

Gaussian elimination with partial pivoting (GE) is perhaps the most well-known direct method for solving a linear system of equations of the form, $Ax = b$, where A is a NxN coefficient matrix, b is a right-hand vector, and x is the solution vector. GE is a floating-point numeric computation that contains two phases, forward reduction and backsubstitution. The forward reduction phase reduces the matrix to an upper-triangular form and is dominant with $O(N^3)$ complexity, while backsubstitution solves the upper-triangular system and has $O(N^2)$ complexity. In the parallel implementation of GE the *PDU* is defined to be a row, and the implementation performs a row-cyclic decomposition of the matrix. Since the amount of computation performed on a row decreases for rows further down in the matrix, a cyclic interleaving of rows gives better load balance. The tasks are arranged in a broadcast topology for pivot exchange.

Experimental Results

STEN

STEN has been run on a range of grid sizes: N= 64, 128, 256, 512, 1024, and 2048, from small- to large-grained, see Table 1. The *configuration* is the number of workstations that Prophet selected from each cluster, *etime* is the reported wall-clock time for the problem instance, *benefit* is the performance improvement achieved by Prophet as compared with the performance that could be obtained by using the best single cluster configuration, and T_c is the per-cycle (per-iteration) execution time. Each data point is the result of five runs with averages presented. The elapsed times do not include Prophet overhead, they are presented in a subsequent table. Other overheads including data distribution and instantiation are also not included since these costs are not induced by Prophet per se. They will be paid by the application whether Prophet is used or not.

A number of comparisons are made to assess solution quality. The *benefit* is computed by comparing the performance obtained by Prophet to the best performance that could be obtained if a single cluster of homogeneous workstations is used. This is determined by running the code for all possible number of workstations within the individual processor clusters until the best was found. For small problems that require only workstations in a single processor cluster, Prophet always finds the best number of workstations. Small problems also highlight the importance of cluster ordering. For larger problems there is a benefit to using heterogeneous workstations in multiple processor clusters even in the presence of conversion and routing overhead. However to exploit heterogeneous workstations in multiple clusters, partitioning and placement must be done carefully.

Prophet predicts the execution time for each candidate schedule (i.e., configuration) by estimating the per-cycle execution time T_c. The accuracy of Prophet is demonstrated as the predicted T_c agrees with the measured T_c within 10%. The results indicate that good performance is obtained, but that the performance improvement depends on problem size. The best sequential times for the fastest CPU type (the SGI) are also included as a reference point. The best sequential time for the small problems outperforms the parallel code since the sequential application uses statically allocated arrays while the parallel code uses dynamic data structures.

The performance benefit depends on two aspects of scheduling, proper data domain decomposition and placement. This dependence is shown for problem instances that require heterogeneous workstations (i.e., processors in multiple clusters). To quantify the dependence on a proper data domain decomposition, an equal share of the data domain has been assigned to each task regardless of its placement (and continued to use automated placement based on topology). To quantify the dependence on placement, a random assignment of tasks to processors ensuring a single task per processor has been performed (and continued to use a proper data domain decomposition). The results are summarized in Table 2. An equal decomposition of the data domain causes a load imbalance and illustrates the importance of Prophet's automated partitioning process. A random task placement creates additional communication overhead and illustrates the importance of Prophet's automated task placement.

Finally, automating the scheduling process has tolerable overhead, see Table 3. Prophet overhead is at most one percent of the total elapsed time (usually less) and is easily amortized. As the problem gets larger, the overhead is more easily tolerated. The cost of endian conversion, the most common form of data format conversion required for heterogeneous workstation networks, has also been considered. For every communicated message, the sending or receiving task performs an endian conversion on the message. The increase in per-cycle execution time T_c due to this additional cost is also easily managed.

FEM

FEM has been run on three instances of the finite-element problem, *dct3*, containing 2160 finite elements, *dcq9*, containing 2304 finite elements, and *dcq9x2*, containing 9216 finite elements. The stiffness matrix sizes are *N*=1117 for *dct3*, *N*=9303 for *dcq9,* and *N*=37057 for *dcq9x2*. These problem instances are very sparse with the average number of non-zeros per row: 10 for *dct3*, 26 for *dcq9*, and 104 for *dcq9x2*. The initial set of results for FEM are presented in Table 4. The solve and assembly phases are handled separately by Prophet.

Prophet produces good performance and accuracy for the assembly and solve data parallel computations. This performance depends on load balance produced by a proper data domain decomposition and task placement, see Table 5. Since assembly uses a broadcast, a random placement is used by the application and no benefit is showed for automated task placement.

Prophet overhead is also manageable for FEM, see Table 6. No endian conversion was used in *dct3-solve* which uses only a single processor and does not require communication.

CL

CL has been run using two different sequence comparison algorithms, Fasta (FA) and Smith-Waterman (SW) on five problem instances. Two used Fasta, *FA-1* and *FA-2*; and three were run using Smith-Waterman, *SW-1*, *SW-2*, and *SW-3*. Smith-Waterman is a much more computationally-intensive comparison algorithm. A problem instance is defined by a particular source library that is decomposed across the CL tasks. In all cases the same target library is used, a library containing 1439 sequences. This corresponds to the number of cycles or iterations. The source library sizes are the following: 287 sequences for *FA-1*, 4397 sequences for *FA-2*, 144 sequences for *SW-1*, 620 sequences for *SW-2*, and 1439 sequences for *SW-3.* All libraries have been randomized to help insure load balance when the source library was distributed across the tasks. The initial set of results for CL are presented in Table 7.

The large improvement result for CL is due, in part, to the loosely-coupled structure of this code and the large computation granularity inherent in the problem instances. As before, the performance improvement depends on the data domain decomposition and task placement, see Table

8. The benefit of a proper data domain decomposition is much more important than a proper task placement due to the large computation granularity.

Prophet overhead is also manageable for CL, see Table 9. The Prophet overhead is a little higher than for STEN or FEM since a larger set of configurations are explored due to the large computation granularity for the CL problem instances.

GE

GE has been run on a range of matrix sizes: N= 256, 512, 768, 1024, and 2048, from small- to large-grained, see Table 10. GE scales very poorly on the network due to the substantial amount of global communication via broadcast and Prophet is only able to use processors from C_1. For this reason the performance benefit is due only to a proper processor selection, both in type (the faster SGI's), and in number. The overheads for GE are similar to the other applications and are omitted.

The results indicate that the precise costs or benefits experienced for all applications are both problem and environment dependent. Different problem sizes may exhibit different performance behavior due to memory and cache effects. For very large problems it is possible that paging also had an impact. However the suite of applications and problem instances were varied enough to suggest several trends in the experimental results. Prophet overhead and the cost of endian conversion is on the order of a few milliseconds for all applications. The performance benefit obtained by Prophet over the single fastest cluster (SGI's) ranged from 10-40% with a much higher benefit over the two slower clusters (Sparc2's and IPC's). This benefit was generally higher for problem instances with larger computation granularity. The performance benefit of a proper data domain decomposition was close to 100% for large problems and between 10-30% for smaller problems. The benefit of automated placement ranged from 30-75%, but depends on the communication topology and the computation granularity. For example, STEN is a fairly tightly-coupled code that benefits a great deal by placement, around 75%, while CL is more loosely-coupled and the benefits are more modest, closer to 30%.

6.0 Summary

Many parallel processing systems for heterogeneous workstation networks are limited in their support for application scheduling. Most users are required to hand-schedule their application, often specifying the number and type of workstations to use. However, it is possible to automate this complex process providing that information is made available to a scheduling system. In this sense Prophet serves as a proof-of-concept for SPMD applications. The Prophet architecture could be implemented in other parallel processing systems improving their functionality and performance. Prophet is currently being extended in several directions, support for greater heterogeneity including multiprocessor workstations and different LAN technologies including ATM and Myrinet, support for dynamic load balancing, and scheduling in a wide-area environments.

7.0 References

[1] M.J. Atallah, et al, "Models and algorithms for co-scheduling compute-intensive tasks on a network of workstations", *Journal of Parallel and Distributed Computing*, Vol. 16, 1992.

[2] F. Berman et al, "Application-level Scheduling on Distributed Heterogeneous Networks," *Proceedings of Supercomputing* 1996.

[3] N. Carriero et al, "Adaptive Parallelism with Piranha," Technical Report #954, Yale University, February 1993.

[4] J. Casas et al, "Adaptive Load Migration systems for PVM," *Supercomputing* 1994.

[5] M. Colajanni and M. Cermele, "DAME: An Environment for Preserving the Efficiency on Data-Parallel Computations on Distributed Systems," *IEEE Concurrency*, January 1997.

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

[7] P.J. Hatcher, M.J. Quinn, and A.J. Lapadula, "Data-parallel Programming on MIMD Computers," *IEEE Transactions on Parallel and Distributed Systems*, Vol 2, July 1991.

[8] N. Nedeljkovic, and M.J. Quinn, "Data-Parallel Programming on a Network of Heteroge-

neous Workstations," *Proceedings of the IEEE First Symposium on High Performance Distributed Computing*, Sept. 1992.

[9] W. Shu and L.V. Kale, "Chare Kernel - a Runtime Support System for Parallel Computations," *Journal of Parallel and Distributed Computing,* Vol. 11, 1991.

[10] V.S. Sunderam, "PVM: A framework for parallel distributed computing," *Concurrency: Practice and Experience*, Vol. 2(4), December, 1990.

[11] 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.

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

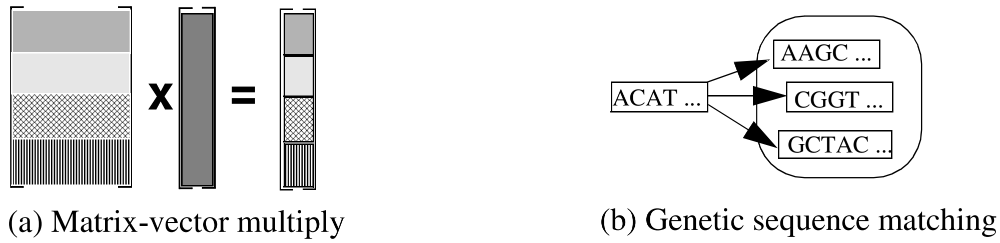

Figure 1: Larger-grain data parallel problems

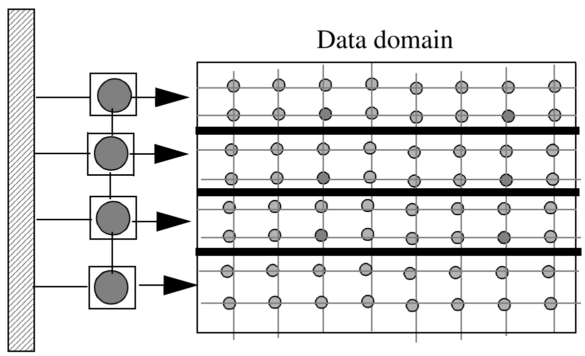

Figure 2: SPMD application. The problem is decomposed into a set of data regions (rectangles) each assigned to a task (large circle) which is placed on a processor (square). The tasks are arranged in a 1-D communication topology.

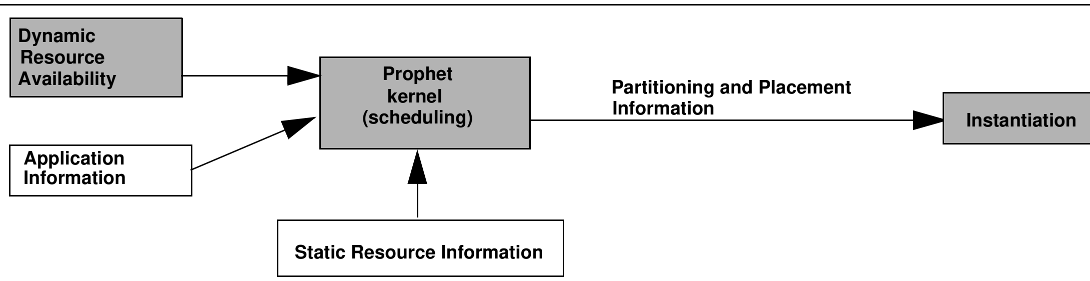

Figure 3: Prophet system architecture

topology ⇒ 1-D
comm_complexity ⇒ 4N (bytes)
numPDUs ⇒ N
comp_complexity ⇒ 5N (fp ops)
arch_cost ⇒ SGI: = .0001
⇒ Sparc2: = .000319
⇒ IPC: = .0006

Figure 4: Example: callbacks for 1-D stencil

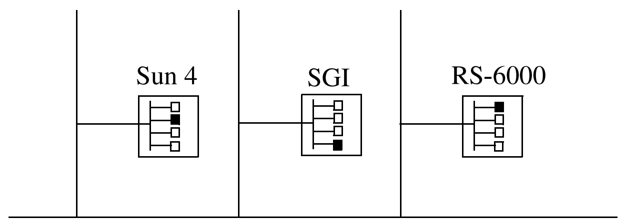

Figure 5: NOW Network organization

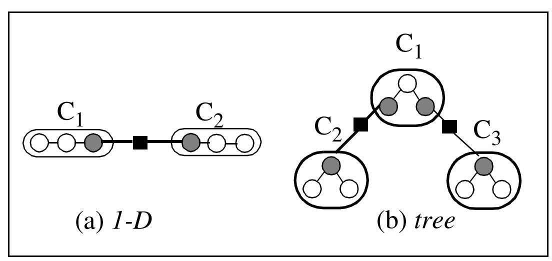

Figure 6: Placement

```
class domain {
char** PV;
public:
    virtual domain (char** curr_PV);
    virtual phase dominant_comp_phase (int np)=0;
    virtual phase dominant_comm_phase (int np)=0;
    virtual phase_rec num_phases ()=0;

    virtual comp_rec comp_complexity (int np, phase comp_phase)=0;
    virtual int numPDUs (phase comp_phase)=0;
    virtual cost_rec arch_cost (host_types proc, phase comm_phase)=0;

    virtual comm_rec comm_complexity (int np, phase comm_phase)=0;
    virtual top topology (phase comm_phase)=0;
};
```

Figure 7: Callback interface

```
class stencil_domain : domain {
public:
  ...
  comp_rec comp_complexity (int np, phase comp_phase) {
    int N = atoi (PV[0]); // extract problem size
    comp_rec CR;
    switch (comp_phase) { // only one comp phase
     case 0 :
       CR.PDU_inst = 5*N; // 5 fp operations per PDU in this problem
       break;
    }
    return CR;
  }
  ...
```

Figure 8: Implementation of stencil callbacks

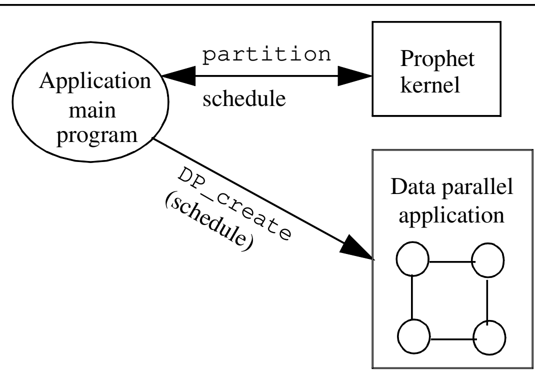

Figure 9: Instantiation Process

```
void main() {
    partition_rec *PR;
    sten_worker *objects, mo;
    stencil_domain *dom;
    DD_floatarray *Grid;
    ...
    // Problem-specific code: (N, iters, and Grid are read from file)
    PV[0]= itoa (N); // marshal PV for problem instance
    dom = new stencil_domain (PV); // instantiate domain

    PR = partition (dom);
    objects = (sten_worker*) DP_create (PR, mo);

    // Application-specific code
    1D_grid = 1D_carve (Grid, PR.partition_map);
    for (int i=0; i<PR.total; i++)
        objects[i].init_grid (1D_grid[i], N);
    for (int i=0; i<PR.total; i++)
        objects[j].compute_grid (iters);
    ...
}
```

Figure 10: Stencil main program

Problem Size	**Seq time (msec)**	**Configuration**			**Etime (msec)**	**% Benefit using selected configuration**			**T_c (msec/cycle)**	
		C_1	C_2	C_3		C_1	C_2	C_3	predicted	actual
64	174	1	0	0	200	---	178%	481%	2.1	2.0
128	698	3	0	0	776	---	53%	216%	7.3	7.8
256	2924	4	0	0	1620	---	44%	176%	16.1	16.3
512	13287	6	8	0	4390	9%	52%	159%	38.4	43.9
1024	51550	6	8	5	9635	25%	139%	396%	95.1	96.4
2048	282984	6	8	6	36558	38%	122%	478%	346.8	365.6

Table 1: Experimental results for STEN

Problem Size	**Equal decomposition Etime (msec)**	**% Increase over Prophet**	**Random placement Etime (msec)**	**% Increase over Prophet**
512	5125	17%	6483	48%
1024	18201	89%	17842	76%
2048	64903	77%	63628	74%

Table 2: Benefit of automating partitioning and placement for STEN

Problem Size	Prophet overhead (msec)	Endian conversion Etime (msec)	% increase in T_c
64	7.3	200	---
128	6.8	790	1%
256	7.2	1649	1%
512	10.5	4579	4%
1024	10.2	10386	8%
2048	10.7	38882	6%

Table 3: Overhead for STEN

Problem	Seq time (msec)	Configuration			Etime (msec)	% Benefit using selected configuration			T_c (msec/cycle)	
		C_1	C_2	C_3		C_1	C_2	C_3	predicted	actual
dct3-assembly	1383	4	0	0	1153	--	27%	78%	49.4	53.4
dct3-solve	2531	1	0	0	2913	---	125%	296%	28.1	27.8
dc9q-assembly	13422	6	4	0	2905	16%	59%	253%	134.8	126.1
dcq9-solve	73712	4	0	0	48410	---	80%	305%	115.5	124.4
dcq9x2-assembly	53544	6	4	0	9660	17%	47%	254%	103.2	105.0
dcq9x2-solve	1642404	6	8	0	272079	12%	111%	274%	676.7	701.0

Table 4: Experimental results for FEM

Problem Size	Equal decomposition Etime (msec)	% Increase over Prophet	Random placement Etime (msec)	% Increase over Prophet
dc9q-assembly	4862	67%	---	---
dcq9x2-assembly	12916	34%	---	---
dcq9x2-solve	536023	94%	361840	31%

Table 5: Benefit of automating partitioning and placement for FEM

Problem Size	Prophet overhead (msec)	Endian conversion Etime (msec)	% increase in T_c
dct3-assembly	10.8	1195	4%
dct3-solve	10.5	---	---
dc9q-assembly	10.1	3279	6%
dcq9-solve	13.4	49152	2%
dcq9x2-assembly	7.8	10061	4%
dcq9x2-solve	15.3	275064	1%

Table 6: Overhead for FEM

Problem	Seq time (msec)	Configuration			Etime (sec)	% Benefit using selected configuration			T_c (msec/cycle)	
		C_1	C_2	C_3		C_1	C_2	C_3	predicted	actual
FA-1	372	4	0	0	151	---	157%	409%	100.6	105.4
FA-2	5695	6	5	0	1637	3%	164%	371%	1152.1	1137.7
SW-1	18649	6	0	0	3646	---	238%	722%	2438.4	2533.9
SW-2	80296	6	8	0	11594	40%	527%	1102%	7513.8	8057.3
SW-3	185069	6	8	8	22807	32%	457%	964%	15618.4	15849.5

Table 7: Experimental results for CL

Problem Size	Equal decomposition Etime (msec)	% Increase over Prophet	Random placement Etime (msec)	% Increase over Prophet
FA-2	3297	101%	2130	30%
SW-2	23573	103%	14985	24%
SW-3	42996	89%	29100	28%

Table 8: Benefit of automating partitioning and placement for CL

Problem Size	Prophet overhead (msec)	Endian conversion Etime (msec)	% increase in T_c
FA-1	12.1	155	2%
FA-2	14.5	1656	1%
SW-1	11.2	3673	1%
SW-2	15.7	12048	4%
SW-3	15.6	22800	0%

Table 9: Overhead for CL

Problem Size	**Seq time (msec)**	**Configuration**			**Etime (msec)**	**% Benefit using selected configuration**			**T_c (msec/cycle)**	
		C_1	C_2	C_3		C_1	C_2	C_3	predicted	actual
256	1743	1	0	0	1504	---	151%	322%	5.7	5.9
512	13900	2	0	0	8891	---	35%	205%	16.2	17.4
768	50524	3	0	0	21783	---	84%	197%	26.3	28.4
1024	123053	4	0	0	40817	---	66%	227%	37.9	39.9
2048	1089355	6	0	0	259150	---	102%	231%	118.4	126.6

Table 10: Experimental results for GE