Gallop: The Benefits of Wide-Area Computing for Parallel Processing*

Jon B. Weissman

Assistant Professor of Computer Science
Division of Computer Science
University of Texas at San Antonio

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

Jon B. Weissman

Division of Computer Science
University of Texas at San Antonio
6900 North Loop 1604 West
San Antonio, TX 78249
(jon@cs.utsa.edu)

Abstract

Internet computing has emerged as an attractive paradigm for many applications that require access to distributed resources such as telemedicine, collaboratories, and transaction systems. The performance of these applications is often secondary to the convenience, functionality, and availability offered by distributed computing. However, it is possible to obtain high performance for a different class of applications, parallel scientific applications. A wide-area scheduling system called Gallop has been developed to exploit opportunities for high performance using remote Internet resources for these applications. This paper describes the Gallop architecture and scheduling model, and performance results obtained for three parallel applications. The initial results indicate that wide-area parallel processing can lead to better performance even with current Internet technology.

Symbols

s_i - site i

$[b_1, b_2]$ - bandwidth range

T_{CT} - completion time

$\overline{T_{comm}}$ - communication time

$\overline{T_{comp}}$ – computation time

k - number of computation stages

n - number of pipelines

N - problem size

c_1, c_2 - communication constants

p - number of processors

bw_avail - available bandwidth

cpu_avail - available CPU

arch_cost - architecture execution cost

comp_complexity - computation complexity

comm_complexity - communication complexity

cycles - number of iterations

num_inputs - number of inputs

numPDUs - number of primitive data units

1.0 Introduction

Internet computing is being successfully applied to an increasing number of applications. Much of the driving force behind the current interest in Internet applications is the development of distributed software technology, most notably Java and the World-Wide-Web. The vast majority of Internet applications including telemedicine, collaboratories, transaction systems, information browsing and collecting require predictable "real-time" performance. For applications that require high performance, the Internet is assumed to be inappropriate. However, it is possible to achieve high performance for computationally-demanding applications in a wide-area environment. High performance can be achieved in one of two ways: *metacomputing* and *remote computing*. Metacomputing aggregates the computation, memory, disk, and communication resources of multiple wide-area sites providing the illusion of a single, powerful, *metacomputer* to the user. For the metacomputing paradigm to be feasible, high-speed wide-area networks must connect the sites. Small-scale OC-3 and DS-3 speed wide-area networks are emerging to support metacomputing, but Internet-wide metacomputing is not yet practical for most applications given current Internet communication performance. Remote computing is a special case of metacomputing in which the heterogeneity, ubiquity, and availability of Internet resources can be exploited to locate the best *intra-site* resources for user applications. Remote computing selects a single remote site for application execution and can be beneficial even with current Internet technology. A software system called Gallop has been developed to exploit this opportunity.

Gallop provides a metacomputing abstraction to the user, but restricts application scheduling to run in the best site in a wide-area network. Gallop schedules parallel applications including data parallel (SPMD) and task parallel pipelines (PP) to reduce application completion time. Gallop is built upon two established software technologies, Mentat [9] and Prophet [21]. Mentat provides the programming environment for Gallop applications and the necessary run-time support for remote execution. Prophet provides local site scheduling for parallel applications in shared, heterogeneous, workstation networks. Prophet runs in each wide-area site and is used by the global Gallop scheduler.

The preliminary results are very promising. Gallop was able to exploit opportunities for better performance using remote Internet sites for applications with a wide range of granularities. Gallop is careful to avoid costly scheduling overhead in order to obtain this performance benefit. The results also indicate that current Internet bandwidth is a limiting factor for some applications. However, the emergence of high speed wide-area networks promises to remove this obstacle.

The Gallop parallel application model is described in Section 2.0. It specifies the information needed to select the best site for the application and to remotely execute the application, if necessary. Next, the Gallop wide-area computing model is described in Section 3.0. This model includes the Gallop network architecture and the application scheduling algorithms. Finally, experimental results for parallel applications executed in the Gallop network testbed are presented in Section 4.0. The paper concludes with related work and summary.

2.0 Parallel Application Model

The Gallop scheduling system assumes a parallel application model that describes the application structure and important cost parameters. In this model, the parallel application is assumed to be decomposed into a set of tasks that compute and communicate over the course of the application. This application model is a variant of the static-task graph (STG) model [3][13][17]. Scheduling the application requires two important types of information — *application execution information* (AEI) and *application structure information* (ASI). Application execution information describes the run-time execution structure of the application. This information is used by the system to support application execution in any site. Application structure information quantitatively describes the computation and communication cost structure of the parallel application. This information is used to evaluate candidate resources offered by the local and remote sites. The ASI model depends on the *structure* of the implementation, but it does not assume a particular language or system implementation. It could be used to describe a PVM, Mentat, DataParallel C, or other parallel application. In contrast, the AEI model is specific to a particular language implementation. An AEI for SPMD and task parallel applications written in Mentat has been implemented in Gallop.

Mentat is an object-oriented parallel processing system based on C++ suitable for medium- to coarse-grain parallel applications [9]. 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, called Mentat objects, are implemented by address-space disjoint processes, and communicate via remote methods. Parallelism is expressed in Mentat by defining a Mentat class that corresponds to a computation task and instantiating some number of Mentat objects of this class. Separate Mentat objects may run in parallel, and Mentat class methods may in-turn be implemented in parallel via other Mentat objects. Mentat has been developed for LAN environments and parallel computers and has recently been extended to support wide-area computing. These extensions are the precursor to a new wide-area system called Legion [11]. Gallop uses one of these extensions, the Legion global name space, to allow sites to access remote application binaries and data files needed to support remote execution.

The Mentat AEI consists of a front-end program, a master or main program, and one or more slave programs all written in MPL (Figure 1). The front-end is responsible for constructing a list of application execution parameters and submitting the application for execution. The front-end is run by the end-user on a local computer. An example front-end is given in Appendix A. The main program is responsible for instantiating the parallel computation represented by the slave programs, and returning results to the front-end program. Once a scheduling decision is made, the

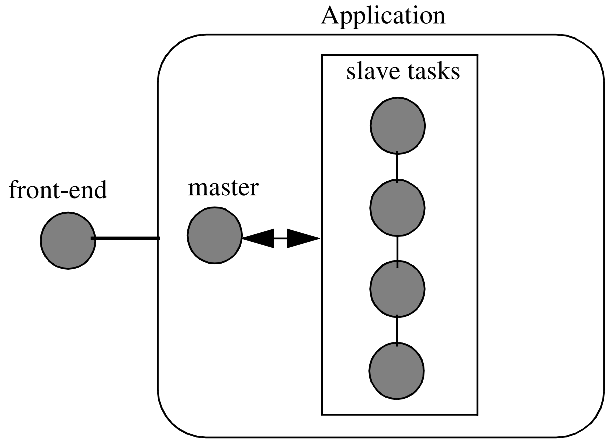

Figure 1: Application execution structure

main program and the slave programs are run in the same site. The main program is encapsulated by a Mentat class `main_prog_class` (Figure 2). The main program will be "launched" by creating a Mentat object of this class on a computer in the selected site. Currently, the `main_prog_class` is used to encapsulate Mentat programs. In the future, it will be used to wrap applications written in other languages such as PVM. The application execution parameters assembled by the front-end include the following:

- main program class file -- file name of `main_prog_class` implementation
- slave binary files -- binary file names for slave programs
- command-line arguments -- any parameters needed by the application
- input files -- input files needed by the application
- output files -- output files produced by the application
- constraints -- see below
- execution history -- see below

The only mandatory parameter is the main program class file. If a remote site is selected, the system will transport input files, output files, and binaries, if necessary. Issues to relating to binary recompilation and security, while important, are outside the scope of this paper. Constraints include application execution dependencies such as the application is written in PVM and requires a PVM-aware site, or application resource dependencies such as the application requires a specific type of computer, or application deadline requirements such as the application must finish in under *t* time units. Supporting constraints is the subject of future work. Maintaining applica-

```
persistent mentat class main_prog_class
{
public:
  // Launch application using schedule s, with arg_list A
  // arg_list contains any command-line arguments
  float launch (schedule *s, arg_list *A);

  // Terminate application
 void teardown ();
};
```

Figure 2: Main_prog_class specification

tion execution history is also a subject of future work. History information can be used to select or avoid sites based on prior performance.

An ASI model for two classes of tightly-coupled parallel applications, medium- to coarse-grain data parallel (SPMD) and task parallel pipelines (PP) has been completed[†]. In the ASI model, parallel applications consist of computation and communication phases. Typically these phases are executed repeatedly for some number of iterations or cycles. 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 SPMD application. In a PP image-processing application, it might be an input image. Information about the computation and communication phases is provided by a set of functions called *callbacks*. For each computation phase, the following information must be provided by the callbacks:

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

†. A third class called meta-applications (MA) which consist of large-grained loosely-coupled components is the subject of future work.

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 machine type in the network. To obtain the *arch_cost*, the sequential code must be benchmarked on each machine type. For each communication phase, the following information must be provided by the callbacks:

- *topology*
- *comm_complexity*

The topology refers to the communication pattern of the application, e.g., *1-D*, *ring*, *tree*, etc. The amount of communication between application 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 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. For example, the callbacks for an SPMD application that uses a five-point stencil to compute poissons equation on a *N*x*N* grid is shown in Figure 3 (the *arch_cost* is omitted). This implementation uses a row decom-

topology ⇒ 1-D
comm_complexity ⇒ 4N (bytes)
numPDUs ⇒ N (rows)
comp_complexity ⇒ 5N (fp ops)

Figure 3: Callbacks for 1-D stencil

position of the grid with the constituent tasks communicating in a *1-D* pattern. The *PDU* is a row of the grid. This application consists of a dominant computation phase where each task updates its grid points in parallel, and a dominant communication phase where each task transmits a border row containing 4-byte grid points, to its north and south neighbors.

The ASI callback information is contained in a C++ class called a *domain* (Figure 5). The domain class consists of a set of member functions that correspond to the callbacks. This class must be derived and tailored for each parallel application. However, a library of classes for common application types such as stencil problems can be easily developed to simplify this task for the application programmer. The domain class is used by the Prophet scheduling system [21], a software scheduler that runs at every site described in Section 3.3.

The AEI and ASI parameters are encapsulated by a class `Job_class` (Figure 4). The `Job_class` is a Mentat class – an object of this class is created to run locally by the front-end and accessed by Gallop. The contained information will be used by Gallop as described in the next section.

```
class domain {
    char** PV; // PV contains problem parameters needed by the callbacks
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 comp_phase)=0;

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

Figure 5: Callback interface

```
persistent mentat class Job_class {

public:
// ****************************
  // AEI part of job class
  // ****************************
  arg_list *args;
  main_prog_class main_prog;
  arg_list *get_arg_list ();
  void init_execution (arg_list*);
  sched_hist get_hist (); // future
  sched_constraints get_constraints (); // future

  // ****************************
  // ASI part of job class
  // ****************************
  domain *job_domain;
  domain *get_job_domain ();
};
```

Figure 4: Job_class specification

3.0 Wide-Area Computing Model

3.1 Wide-Area System Architecture

Gallop represents the wide-area network as a collection of autonomous fully-connected sites (Figure 6). A hierarchical scheme may be explored in the future since it is more scalable for a large number of sites, but a flat organization is adequate for modest size testbeds of 10-100 sites. This size system would include all recent metasystem testbeds such as I-Way/Gusto [7], Nile [14], and testbeds that connect the national laboratories, various universities, and corporate enterprises. Furthermore, a flat organization has better fault tolerance properties.

Within a site, resources are organized into *processor clusters* (Figure 7). Processor clusters contain computers on separate subnetworks. Information about a sites computing resources is needed to evaluate potential scheduling decisions. The system maintains the following information about each processor cluster within a site:

- *processors*
- *aggregate power*
- *communication functions*

The *processors* include the number and type of the available computers, and their load status. The *aggregate power* is the total Mflops and Mips available. The *communication functions* predict the communication cost as a function of message size and number of processors for the cluster. Examples of communication functions for workstation-based clusters are provided in Section 3.3.

Each site runs a *scheduling manager* (*SM*) and a *local scheduler* (*LS*). The *SM*'s run the wide-area scheduling algorithm discussed later in Section 3.2. The *SM* interfaces to the site *LS* and to the other *SM*'s. The *LS* is responsible for managing the local site resources in a manner transparent to the *SM*. It is the *LS* that decides what resources the site will make available for wide-area scheduling at any point in time, thus providing site autonomy. The *LS* contains two components. One component is an interface to the *SM* and the other is an interface to a site-specific scheduling system (e.g., NQS, Condor, Prophet, etc.). As an example consider two sites as

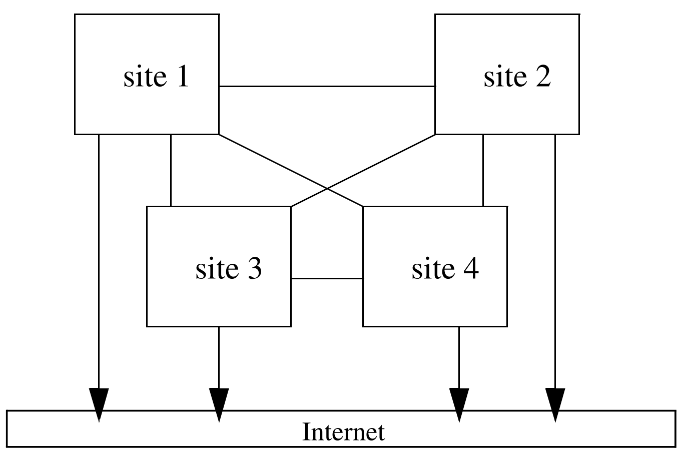

Figure 6: Wide-area system organization

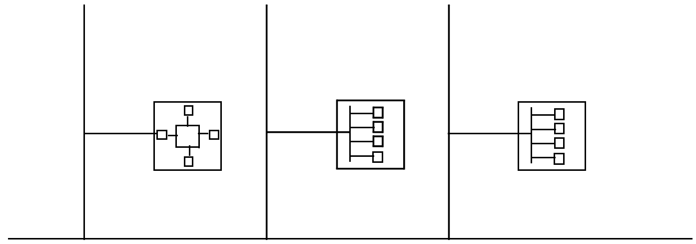

Figure 7: Intra-site organization. A site containing 3 processor clusters, two connected via ethernet and the other via ATM.

depicted in Figure 8; site 1 contains a single parallel supercomputer (perhaps an Intel Paragon or SP-2), and site 2 contains a workstation network. The *LS* for site 1 uses NQS and the *LS* for site 2 uses Condor. NQS manages the supercomputer partitions and Condor manages the workstations. For sites that contain a multitude of resources, several *LS*'s may be used with a single representative designated to communicate with the site *SM*. The *SM*'s and *LS*'s are address-space disjoint, communicate via message-passing, and will run on one of host computers within each site. However, in some systems, the *SM* and *LS* may be configured as a single process. In this system prototype, the *LS* is the Prophet scheduler [21]. The *LS* will explore and evaluate potential resource schedules within the site using application information provided by the callbacks (Figure 5) and the resource information described above.

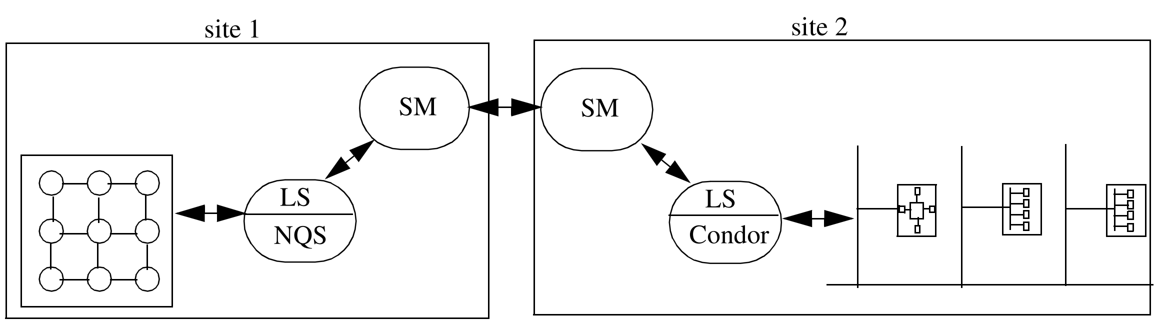

Figure 8: Scheduling components

The current system will schedule an application within a single site. Scheduling an application across multiple sites may be feasible for sites that have a high communication capacity between them (e.g., 45Mb/sec bandwidth or higher). This requires a tighter integration of the site *SM*'s through the *virtual site* mechanism. A virtual site is composed of multiple sites with sufficient communication capacity to be considered as a single site by scheduling. As an example consider the I-Way and Casa virtual sites as depicted in Figure 9 (the virtual sites are the dotted boxes). A single application could be scheduled across the multiple sites connected on the I-Way given this organization. Loosely-coupled meta-applications may be suitable for virtual sites. The implementation of the virtual site mechanism is the subject of future work.

In the current implementation, sites know very little about the resources contained in other sites. A mechanism that allows sites to publish their available resources in terms of intra-site CPU, memory, and network bandwidth is being developed. Currently, each site does maintain inter-site

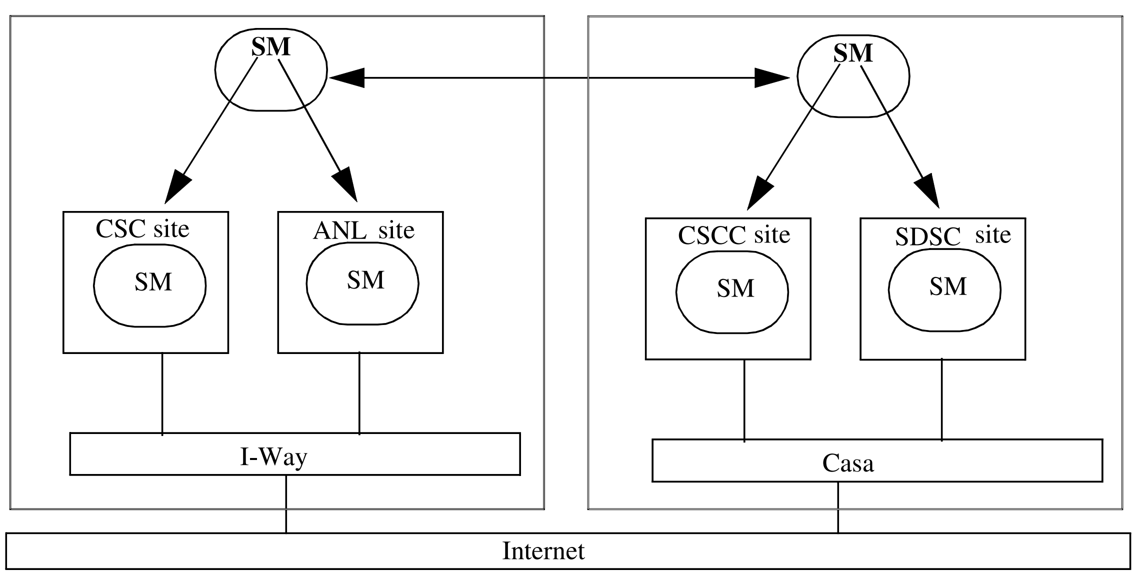

Figure 9: Virtual sites

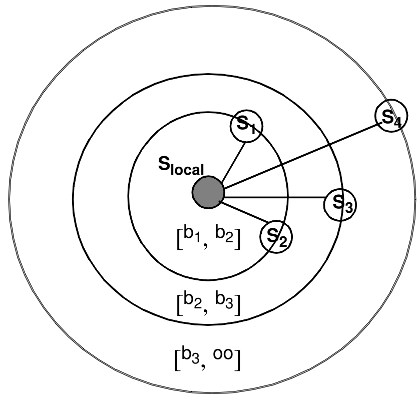

Figure 10: Inter-site communication model

dynamic bandwidth information to all sites via a *spoking* model (Figure 10). Spokes form a bandwidth range. For example, the bandwidth from s_{local} to sites in the first spoke (s_1 and s_2) is between $[b_1, b_2]$. The bandwidth decreases further away from the center. The final spoke represents sites with extremely poor bandwidth or are unreachable. At a preset interval, each site issues a set of network pings to the other sites to determine site availability, inter-site bandwidth and latency. Based on the bandwidth, the remote sites can be placed in their appropriate spoke. At different times of the day, sites may move between spokes as the network load changes. This inter-site communication information is used in estimating scheduling and remote execution overhead, and for site selection, described in Section 3.2.

3.2 Wide-Area Scheduling

The wide-area scheduling process contains a global and local phase. The global phase is initiated by the arrival of a scheduling request to a local site *SM*, produced by the front-end program (Appendix A). The *SM* is passed the `Job_class` object to initiate scheduling of the application. The wide-area scheduling algorithm is a distributed algorithm that has two parts, the local site *SM* component and the remote site *SM* component (Figure 11). The procedure calls across address-spaces (e.g., *get_best*) are RPC's. When an application request first arrives at a local site *SM*, a set of candidate sites are selected (*including* the local site) and the `Job_class` object is sent to each remote site *SM* for *bidding*. Selecting the candidate sites (step 2) is done in the following manner. Sites "closest" to the local site in terms of communication performance are given the greatest priority since files, binaries, results, and scheduling control messages may have to be transmitted between the local and a selected remote site[‡]. The closest sites are located on spokes closest to the local site (Figure 10). For most applications, it is too expensive to consider all available sites. An estimate of the scheduling overhead is used to constrain the number of sites under consideration. This overhead has two terms, protocol overhead (Figure 11), and file transfer overhead in the event that a remote site is selected. The overhead estimate is computed using the spoke latency and bandwidth (for the RPC's in steps 3-6, Figure 11a) to each site for the protocol itself and for file transfer, and the local scheduling overhead (steps 2-3, Figure 11b). Fortunately, the local scheduling overhead is paid once as the sites run this algorithm in parallel. But the communication overhead for the protocol and file transfer dominates.

1. local SM receives *WA_sched* (*Job_class*) request
2. local SM chooses k candidates $SM_1 \ldots SM_k$
3. for each SM_i
    call $best_i = SM_i$ *.get_best* (*Job_class*)
4. *the_best* = min {$best_i$. *elapsed_time*} over all i in $1 \ldots k$
5. for each SM_i
    if (SM_i == *the_best.site*)
        result =SM_i *.go_SM* (*Job_class, the_best*)
    else
        SM_i *.no_go_SM* (*Job_class, the_best*)
6. Return *result* and *scheduling info* to front-end

a) Local SM component

1. remote SM receives *get_best* (*Job_class*) from local SM
2. call *scheds* = *LS.get_scheds* (*Job_class*)
3. call *best* = *LS.eval_scheds* (scheds, *Job_class*)
4. lock *best,* return *best* to local SM
5. if remote SM receives *go_SM* (*Job_class, the_best*)
    result = *LS.go_ls* (*Job_class*, *the_best*)
    return *result* to local SM
else if remote SM receives no*_go_SM* (*Job_class, the_best*)
    store *the_best* in table
6. release lock on *best*

b) Remote SM component

Figure 11: Wide-area scheduling algorithm

‡. History and constraint information will be used first in candidate site selection in the near future.

The number of sites selected is constrained such that the estimated scheduling overhead falls within 10% of the best execution time that can be achieved locally. Sites are selected based on their spoke position from the center outwards, until the predicted overhead reaches this limit. The best local execution time is obtained by having the local *LS* run steps 2 and 3 (Figure 11b). For small applications, perhaps no remote sites will be considered, but for larger applications that run longer, a larger number of remote sites will be selected. Currently, all sites within a spoke are weighted equally since no information about intra-site resources is assumed. When sites publish their available aggregate resources, a mechanism for ordering sites within a spoke can be designed. This version is in development.

Once the candidate set of sites is determined, the algorithm proceeds as follows. Each selected remote *SM* contacts its *LS* to get a list of machines and when they are available. The remote *SM* invokes the *LS* to run a local scheduling algorithm to search this set of machines to evaluate the possible schedules. The best schedule and projected completion time are returned to the local site *SM*. Each *SM* locks this schedule until the scheduling transaction is complete. The best schedule has the smallest predicted completion time. The local site *SM* then selects the best site, and initiates the scheduling of the application on that site. This algorithm offers an opportunity to provide robust application scheduling – instead of a single site selection, the system could redundantly schedule the application in multiple sites for fault tolerance. This possibility is being investigated [23].

When the local *SM* receives all of the bids, it selects the best schedule, and informs all of the remote sites of its decision. This information can be used by the other sites in making future scheduling decisions. For example, the other sites can avoid this site in the short-run since it is already running an application. This would allow a Gallop system to implement load-sharing, if desired. At present, all scheduling decisions are based on what is best for the current application. The chosen site *SM* passes the application (i.e., the `Job_class` object) to the local *LS* for execution. Execution is accomplished by the *LS* which creates and launches the `main_prog_class` object. At this point, the other remote *SM*'s can release the lock on their best candidate schedule. The interactions between the various system components are depicted in Figure 12 (site SM_2 is selected to run the application). Additional details may be found in [19].

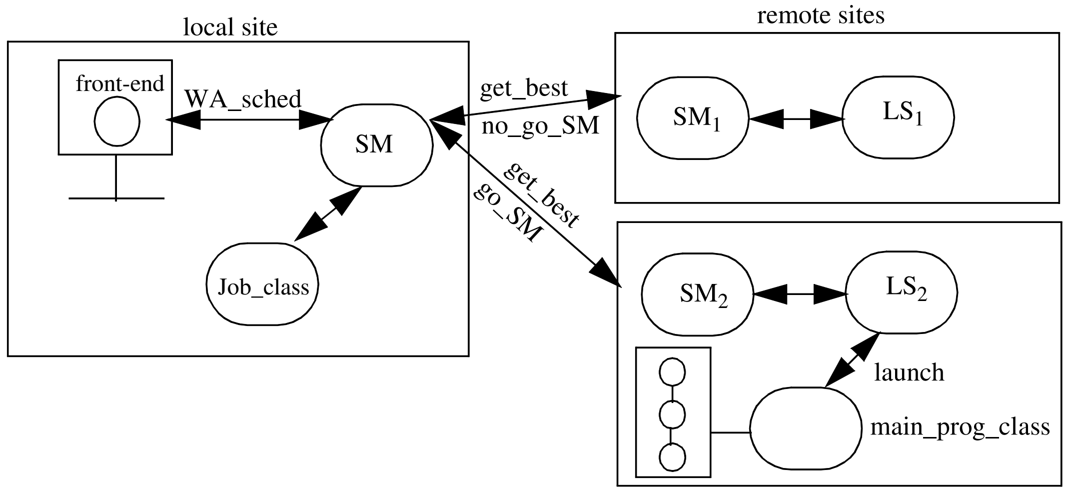

Figure 12: Wide-area scheduling and execution

3.3 Local Scheduling

The Prophet *LS* utilizes a cost-driven method for scheduling SPMD and PP applications in heterogeneous workstation networks. Prophet makes resource scheduling decisions that are predicted to produce reduced completion time for the application. Near-optimal scheduling results have been obtained for SPMD applications [22]. Scheduling SPMD applications consists of two parts, partitioning and placement (Figure 13). In partitioning, the application is decomposed across a set of potentially heterogeneous processors. The number of processors selected depends on the computation granularity — an application with a large computation granularity may benefit by a large number of processors relative to a smaller one. Another factor in processor selection is

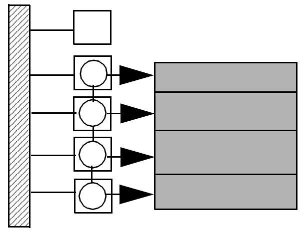

Figure 13: SPMD scheduling. The application is decomposed into a set of data regions (shaded rectangles) each assigned to a task (circle) which is placed on a processor (square). In this example, the tasks are arranged in a 1-D communication topology.

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. A number of *PDU*s will be assigned to each task proportional to the power of the processor to which it is assigned.

For PP applications, a number of inputs are pumped into a pipeline with stages overlapped in execution. The archetype example is an image processing application (Figure 14). The speedup is limited by the pipeline depth and maximum computation time of the pipeline stages. To overcome

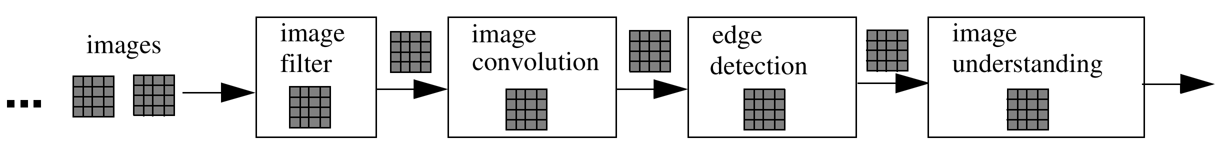

Figure 14: Image processing pipeline

this performance limitation, the pipeline can be replicated and run in parallel (Figure 15). 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. Unlike SPMD applications, the computation tasks may be different. In both PP and SPMD applications, a single computation task is assigned per processor.

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). 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.

Prophet exploits information about the parallel application and the network resources in order to make scheduling decisions for SPMD and PP applications (Figure 16). Prophet explores a set of candidate processor configurations to apply to the application in order to minimize completion time. The details of this algorithm may be found in [21]. Prophet first orders the processor clusters

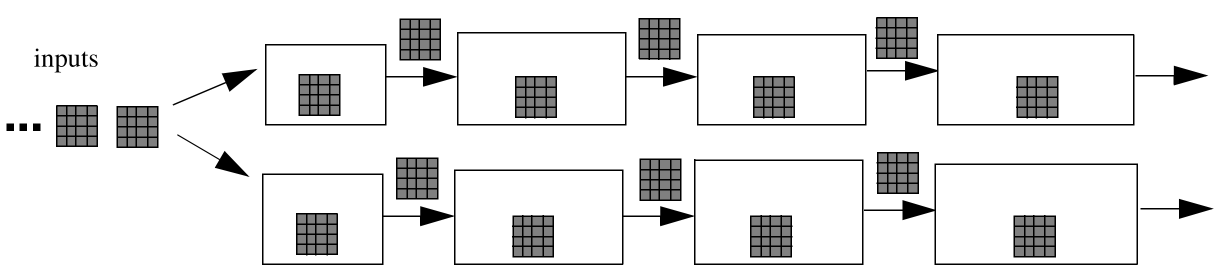

Figure 15: Parallel pipelines

based upon their computation and communication capabilities for the application in order to constrain the search process. The scheduling algorithm estimates the computation and communication rates using the cost functions given below:

$$T_{CT}[SPMD] = cycles \cdot (\overline{T_{comm}} + \overline{T_{comp}})$$

$$T_{CT}[PP] = num\_inputs \cdot k \cdot (\overline{T_{comm}} + \overline{T_{comp}})$$

$$\overline{T_{comp}}[PP] = \frac{(\overline{comp\_complexity} \cdot \overline{arch\_cost})}{k \cdot n \cdot \overline{cpu\_avail}}$$

$$\overline{T_{comp}}[SPMD] = \frac{comp\_complexity \cdot \overline{arch\_cost} \cdot numPDUs}{p \cdot \overline{cpu\_avail}}$$

$$\overline{T_{comm}}[SPMD] = (c_1 \cdot p) + \frac{(c_2 \cdot p \cdot comm\_complexity)}{bw\_avail}$$

Prophet constructs a cost function for application completion time T_{CT} for each candidate schedule**. 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. 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. The term bw_avail is the amount of bandwidth available within the cluster as a percentage (idle network = 100% available). $\overline{T_{comm}}$ is the average communication time spent by a processor or task in one iteration of execution. The communication function shown is for SPMD applications on ethernet which is linear in the number of processors, p, and message size, given by *comm_complexity*. 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

**. The algorithm tries to minimize the sum of T_{comm} and T_{comp}, hence *cycles* is not required for T_{CT} [SPMD].

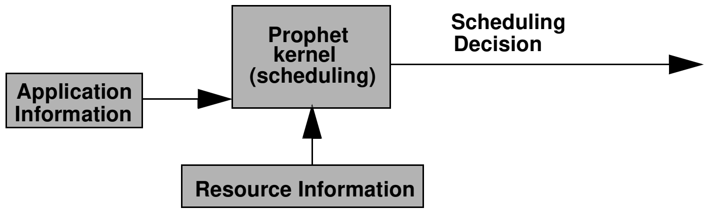

Figure 16: Prophet system architecture

constant. Since the communication parameters reflect the peak communication rates, *bw_avail* is used to scale the bandwidth-dependent portion of $\overline{T_{comm}}$. The communication function for PP applications is more complex since communications from different pipelines will contend for network bandwidth and is omitted. A model for PP communications is provided in [19].

4.0 Results

An experimental Internet-based wide-area testbed containing sites at the University of Virginia (UVA), University of Texas at San Antonio (UTSA), Southwest Research Institute (SWRI) in San Antonio, and Sandia National Laboratory in New Mexico (HEAT) was used to demonstrate the benefits of Gallop. In the testbed, each site contained four Sparc workstations of different power and network connection: UTSA contained four Sun SPARCstation-5s connected on 10 base-T ethernet, UVA contained four Sun SPARCServers-20/514s connected on 10 base-T ethernet, SWRI contained four Sun SPARCstation-5s connected by a Fore ATM SX-100 switch, and HEAT contained four Sun SPARCstation-10s connected by FDDI. All machines are running Solaris 2.x. An important property of this testbed is that the site resources are all similar in computation power. If a remote site contained a supercomputer and the local site contained only workstations then it would be easy to show a benefit to wide-area computing. A more interesting question is whether there is a benefit in the common case — sites containing workstation-based resources.

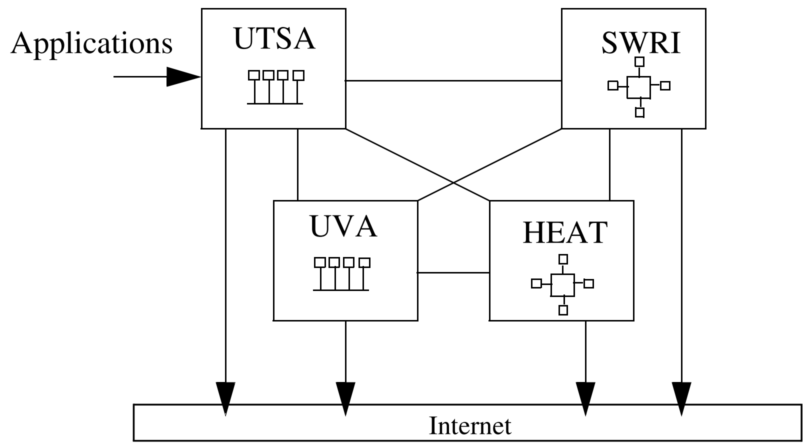

Figure 17: Experimental testbed

The local site is UTSA and all applications are submitted to the *sm* in the UTSA site for execution (Figure 17). The Gallop software including Mentat and Prophet has been installed at each of these sites.

Since Internet communications were expected to dominate the scheduling overhead, a study of Internet communication was performed. The internet communication latency and bandwidth from UTSA to all other sites was recorded over a two week period (Figure 18). The bandwidth data indicates that different sites will be better (i.e., have higher bandwidth) at different periods of the day. The average latency data indicates that the sites have a consistent latency relationship, but a comparison of the average minimum and maximum latency to all sites show a relationship similar to bandwidth. These results suggest that it is not possible to a-priori order the remote sites, rather, a dynamic approach such as the spoking model is needed to most effectively exploit the remote sites.

Three parallel applications have been run in the testbed using Gallop: a five-point iterative solver for poisson's equation (STEN), a parallel image processing pipeline (PP), and a parallel gene sequence comparison code (CL) [10], all written in MPL. STEN and PP are more fully described in [19]. STEN and CL are classic SPMD applications. For each application, different

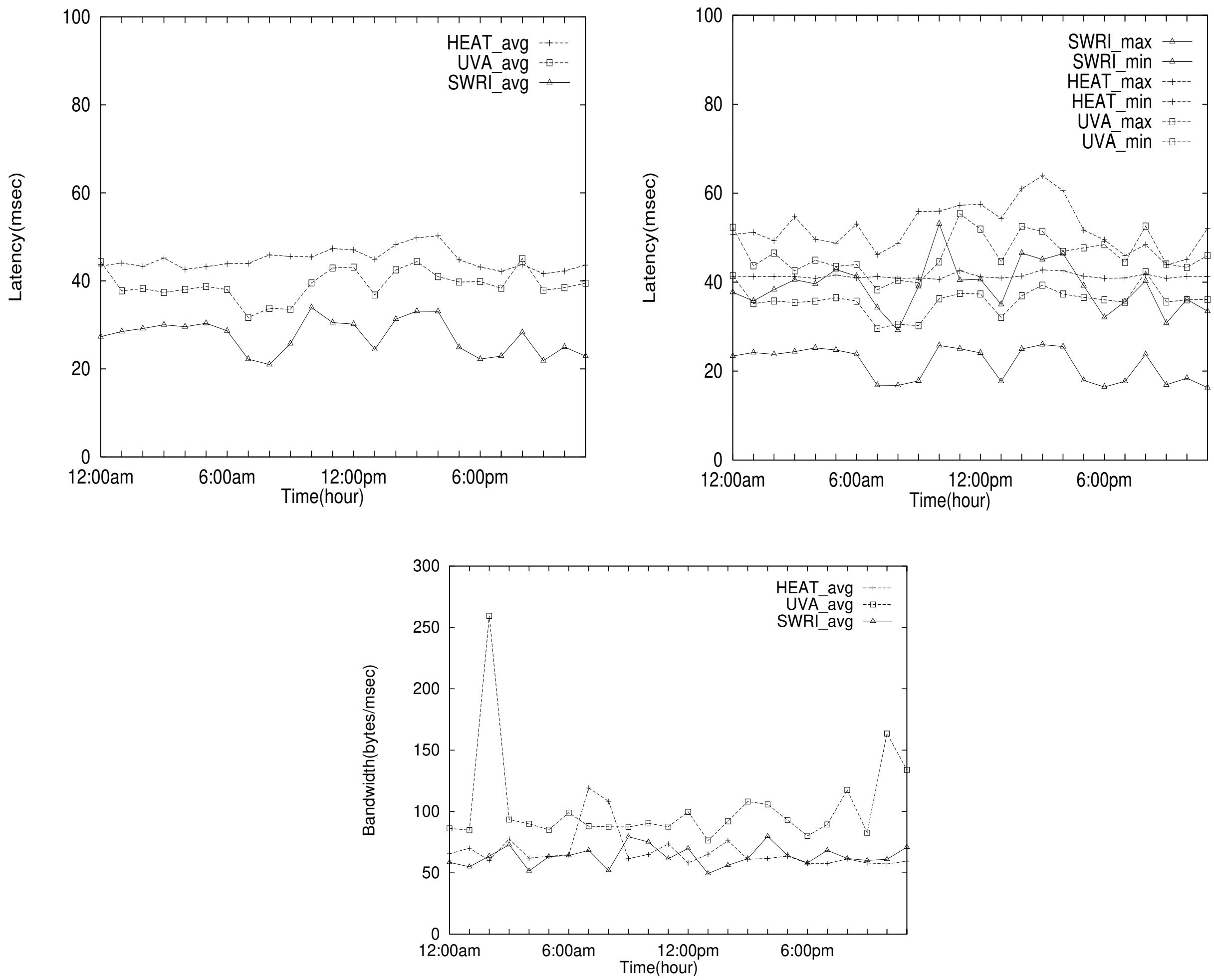

Figure 18: Gallop testbed communication performance

problem instances with different granularities have been run (Table I). For STEN, 100 iterations were run for every problem instance and for PP, 10 input images and a 3-stage pipeline were used for every problem instance. Each problem instance was run approximately 50 times over a 1 week period. The first graph shows the scheduling overhead experienced as a function of the number of sites under consideration averaged over all applications and all runs (Figure 19). It includes all scheduling control overhead, local scheduling overhead, and communications. It does not include the cost to transfer files and binaries to remote sites. The results indicates that overhead is linear in

Table I: Problem instances

Problem instance	**Description**
PP-1	a 100x100 pixel image with 3x3 convolution mask per stage each applied once
PP-2	a 400x400 pixel image with 3x3 convolution mask per stage each applied once
PP-3	a 400x400 pixel image with 3x3 convolution mask per stage each applied three times
STEN-1	a 100x100 grid
STEN-2	a 200x200 grid
STEN-3	a 400x400 grid
STEN-4	a 800x800 grid
STEN-5	a 1024x1024 grid
CL-1	source and target library of 300 sequences each; comparison using FASTA heuristic
CL-2	source and target library of 500 sequences each; comparison using FASTA heuristic
CL-3	source and target library of 100 sequences each; comparison using SMITH-WATERMAN heuristic

the number of sites under consideration. Hence, our algorithm to limit the number of candidate sites is important to retaining the benefit of wide-area computing.

The average elapsed time for all applications is shown in Table II. Gallop was able to exploit superior performance when available at remote sites for many of the problem instances. Even with the addition of scheduling overhead, better performance was obtained remotely. A surprising result was that for small problems such as STEN-1 and STEN-2 there is a benefit to remote execution. Since STEN-1 was so fine-grained, Gallop actually opted to run the application locally even though it could have been executed in HEAT with slightly better performance. For STEN, the HEAT site with FDDI interconnect was selected for the other problem instances. Surprisingly, the FDDI performance was comparable or superior to ATM accept for STEN-3. For CL, the SWRI site performed best as the SWRI machines were idle for most of the CL runs. For PP, the UVA site performed best since PP is compute-intensive and the UVA SPARCservers are the fastest machines in the Gallop testbed.

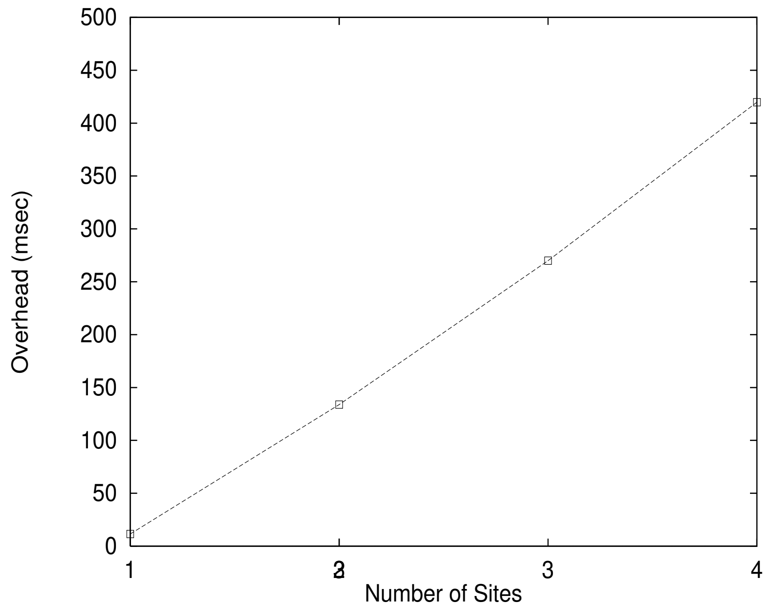

Figure 19: Overhead

The benefit differential between local and remote execution ranged from 100 msec up to 15 sec for these problem instances. While this gap is sufficient to absorb the wide-area scheduling overhead, it is not sufficient to amortize the transmission of binaries and data files for STEN and PP unless the application will be run multiple times. While Gallop will automatically transfer all binaries and data files to a remote site if necessary, all application binaries and data files were pre-installed at all of the sites. Mentat binaries are typically 500 KB in size and would require several seconds to transfer based on measured Internet bandwidths (Figure 18a). The problem is not that the applications are too fined grained to absorb the overhead, but that the gap between local and remote site performance is too small. The reason the gap is relatively small is that the sites are of comparable computation power. As the Gallop testbed grows to include more sites, some including supercomputers, this gap will widen for applications that can utilize supercomputing resources. For these applications, the cost of file and binary transfer and perhaps even remote compilation is much more likely to be amortized. Large scale scientific applications are the most likely candidates for supercomputing resources. Another solution to this problem may be provided by wide-area gigabit networks [4] that would reduce transfer overhead by at least an order of magnitude. This would be sufficient for most of the problem instances presented here. Gallop will be deployed on these future networks as they become available. These networks also make it possible to consider running a single application across multiple sites. The virtual site mechanism (Figure 9) will be used to exploit this opportunity. Current Internet technology limits the class of applications that would benefit by such distribution.

Table II: Application performance. All times shown in milliseconds.

Problem instance	**Gallop schedule**		**Overhead**		**Performance (site elapsed time w/o overhead, total site elapsed time w/ overhead)**			
	Site picked	**# machines**	**UTSA**	**# Sites**	**UTSA**	**UVA**	**SWRI**	**HEAT**
PP-1	SWRI	3	420	4	32785	32726 33146	25819 26239	33999 34419
PP-2	UVA	3	420	4	76020	65958 66378	73210 73630	88768 89188
PP-3	UVA	3	420	4	123774	107890 108310	118195 118615	172819 173239
STEN-1	UTSA	2	12	1	1272	1270 1404	1417 1551	1114 1248
STEN-2	HEAT	4	134	2	3521	5975 6109	2617 2571	2259 2393
STEN-3	HEAT	4	420	4	8775	17297 17717	6970 7390	6304 6723
STEN-4	SWRI	4	420	4	23579	42613 43033	21800 22220	23541 23961
STEN-5	HEAT	4	420	4	42269	56294 56714	39848 40268	37593 38013
CL-1	SWRI	4	420	4	39465	42638 43058	33900 34320	76535 76955
CL-2	SWRI	4	420	4	83479	100645 101065	71838 72258	174099 174519
CL-3	SWRI	4	420	4	80147	200051 200471	78862 79282	331127 331547

5.0 Related Work

A number of other research projects are exploring the benefits of wide-area computing for user applications [1][2][5][7][8][11][14][18]. The Legion project is developing the interfaces and infrastructure for object-oriented wide-area middleware [11]. Since Legion does not mandate specific solutions for problems such as scheduling, Gallop will be integrated into Legion to provide support for scheduling parallel applications. The Globus project is a metacomputing framework designed for applying distributed resources to scientific applications [7]. It relies on a high-speed WAN to interconnect all of the participating sites. Globus is a high-level architecture for constructing metacomputing applications but also lacks scheduling support for parallel applications. A prototype system called GUSTO is in construction. A Gallop GUSTO integration will also be investigated in the future. Infospheres is a framework for building applications that span wide-area systems [5]. It has several components including a distributed directory service needed to locate wide-area resources. Infospheres is targeted to collaborative applications as opposed to high-performance. It does not have any specific support for scheduling or resource management. IceT is a framework for collaborative and high-performance distributed computing based on Java [8]. In IceT, programs and data are moved to the appropriate computational resources. In this respect IceT and Gallop are similar, but Gallop is not tied to a specific programming language such as Java. The Nile project is building a high-performance wide-area computing environment for high energy physics applications. Nile applications number in the thousands per day and may be scheduled on machines located throughout the Nile network. Resource management in Nile is concerned more with job throughput as opposed to application finishing time. The Responsive web computing (RWC) project is exploring the use of the Web as a metacomputer [2]. The primary focus of this work is to reduce the variance of Web computing to support applications with real-time constraints. Specialized TCP protocols and resource management middleware are the key components of the RWC system. However, no specific resource management algorithms have been implemented. Apples is a framework for user-level scheduling of parallel applications in heterogeneous LANs and WANs [1]. 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. Apples is also targeted to LANs primarily. The Apples philosophy is very similar to Prophet which is to use application and resource information to make application-centric scheduling decisions. VDCE is a system for executing tasks across processors that may span multiple sites. The emphasis in VDCE is on the composition of parallel applications from library components.

6.0 Conclusion and Future Work

The Gallop scheduling system shows that high performance and wide-area computing are not incompatible ideas. Gallop was able to exploit opportunities for better performance using remote Internet sites for applications with a range of granularities. Gallop uses resource and application knowledge to avoid costly scheduling overhead when it would not be profitable. The experimental study also revealed that the bandwidth limitation of the current Internet is a problem when large binaries and files must be transported to support remote execution. These costs cannot be absorbed unless the remote site offers an order of magnitude better performance than the local site or the application will be run multiple times. Future work includes deploying Gallop on a fast wide-area network testbed such as GUSTO to overcome this bandwidth bottleneck, and an implementation in the Legion wide-area computing environment [11]. An OC-3 speed network would allow a much wider range of applications to benefit from wide-area execution. An experimental evaluation of Gallop in this environment to test this hypothesis is planned. Future work also includes the development of a scheduling model that would allow single applications to be distributed across multiple sites. The benefit of multiple sites has been demonstrated by large-scale scientific applications [15], but these applications are typically hand-scheduled. Extending Gallop to support the automatic scheduling of applications across multiple sites using application and resource information is planned.

The author would like to acknowledge the following Internet sites that provided resources for our wide-area testbed: University of Virginia Computer Science department, Sandia National Laboratories, and Southwest Research Institute.

7.0 References

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

[2] A. Bestavros, A. et al, "*Responsive Web Computing: Resource Management, Protocol Techniques, and Applications*," Boston University Computer Science Technical Report, 96-008, March 1996.

[3] Bokhari, S.H., Assignment Problems in Parallel and Distributed Computing, Kluwer Academic Publishers, 1987.

[4] Chatterjee, S., "Requirements for Success In Gigabit Networking," *Communcations of the ACM*, Vol. 40(7), July 1997.

[5] Chandy, K.M. et al, "A World-Wide Distributed System Using Java and the Internet," *Proceedings of the Fifth IEEE International on High Performance Distributed Computing* (*HPDC-5*), August 1996.

[6] Eager, D.L, Lazowska, E.D., and Zahorjan, J., "Adaptive Load Sharing in Homogeneous

Distributed Systems," *IEEE Transactions on Software Engineering*, Vol. 12, May 1986.

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

[8] Gray, P.A., and Sunderam, V.S., "IceT: Distributed Computing and Java," *Cluster Computing Conference* (CCC '97), Atlanta, GA, 1997.

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

[10] Grimshaw, A.S., West, E.A., and Pearson W.R., "No Pain and Gain! - Experiences with Mentat on Biological Application," *Concurrency: Practice & Experience*, Vol. 5, issue 4, July, 1993.

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

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

[13] Lo, V.M., "Temporal Communication Graphs: Lamport's Process-Time Graphs Augmented for the Purpose of Mapping and Scheduling," *Journal of Parallel and Distributed Computing*, Vol. 16, 1992.

[14] Marzullo, K. et al, "Nile: Wide-area Computing for High Energy Physics, "*Proceedings of the Seventh ACM SIGOPS European Workshop*, 1996.

[15] Mechoso, C.R. et al, "Running a Climate Model in a Heterogeneous Distributed Computer Environment," *Proceedings of the Third IEEE International on High-Performance Distributed Computing*, August 1994.

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

Performance Distributed Computing, Sept. 1992.

[17] Stone, H.S., "Multiprocessor Scheduling with the Aid of Network Flow Algorithms," *IEEE Transactions on Software Engineering*, Vol. SE-3, No. 1, January 1977.

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

[19] Weissman, J.B., and Zhao, X. "Run-time Support for Scheduling Parallel Applications in Heterogeneous NOWs," *Proceedings of the Sixth IEEE International Symposium on High Performance Distributed Computing*, August 1997.

[20] Weissman, J.B., and Grimshaw, A.S., "A Federated Model for Scheduling in Wide-Area Systems," *Proceedings of the Fifth IEEE International Symposium on High Performance Distributed Computing*, August 1996.

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

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

[23] Weissman, J.B., and Womack, D. "Fault Tolerant Scheduling in Distributed Networks," UTSA Technical Report, CS-96-10, October 1996.

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

Appendix A: Front-end for stencil application

```
void main (int argc, char **argv)
{
  schedule *results;
  sm local_SM;
  stencil_Job_class *j = new stencil_Job_class;
  arg_list *Args;
  program_arg arg;
  arg_type idatafile, odatafile, ldatafile, lbfile;

  local_SM.bind (SEARCH_SUBNET); // locate local scheduling manager
  Args = new (8) arg_list (8);
  // Package up Application Information
  strcpy (arg.arg, "stencil_main_class");
  arg.type = _main_;
  Args->append (&arg);
  strcpy (arg.arg, argv[1]);
  arg.type = _argv_;
  Args->append (&arg);
  strcpy (arg.arg, argv[2]);
  Args->append (&arg);

  // Include grid input file and output file (omitted)
  // ...
  // Include slave binary file
  strcpy (arg.arg, "sten_worker"); // Binaries ...
  arg.type = _bfile_;
  // Register binaries in Legion name space to enable remote copying
  strcpy (lbfile, "/legion/");
  strcat (lbfile, arg.arg);
  lreg_host (arg.arg, lbfile,
             raw_host_id (SELF_ONAME.get_mmps_host()));
  strcpy (arg.arg, lbfile);
  Args->append (&arg);

  j->create (0); // create Job_class object
  if (j->init_domain (Args) == 1); // init domain object
  ((Job_class*)j)->init_execution (Args);
  results = local_SM.WA_sched ((Job_class*)j); // submit application

  // Could log schedule if necessary ....
  fprintf (stderr, "**********FE: stencil took %f msec\n",
   (*results)[0].etime_taken);
  j->destroy ();
}
```