Application Level Scheduling: Gene Sequence Library Comparison*

Neil Spring
Rich Wolski†

July 2, 2002

Abstract

This paper investigates the efficacy of Application-Level Scheduling (AppLeS) **??** *for parallel gene sequencing application in a variety of production metacomputing settings. We compare an AppleS-enhanced version of the application to an original implementation that was designed and tuned to use the native scheduling mechanisms of Mentat* **??** *– a metacomputing software infrastructure. The experimental data shows that the AppLeS versions outperform the best Mentat versions over a wide range of problem sizes and computational settings.*

The structure of the AppLeS we have defined for this application does not depend on the scheduling algorithms that it uses, however, Instead, the AppleS scheduler considers the uncertainty associated with the information it uses to its scheduling decisions to choose between the static placement of computation, and the dynamic assignment of computation during execution. We propose that this framework is general enough to represent the class of metacomputing applications that are organized as a master and set of parallel slaves, in which the master distributes uncomputed work.

1 Introduction

Recent advances in network technology have made it possible for parallel applications to use ensembles of distributed computers to achieve high-performance. Typically, however, it is difficult or impossible to dedicate large-scale, widely dispersed resources to a single application at a single time – applications must be able to execute efficiently using heterogeneous sets of resources that are shared by other users and applications. Since computing in this form involves the interoperation of complete computer systems (including the resident operating systems, languages and language libraries, storage devices, etc. on each) it is often termed *metacomputing* [5, 11, 18, 3].

*Supported by Really Cool People
†email: nspring@cs.ucsd.edu, rich@cs.ucsd.edu

To take the fullest advantage of the shared, heterogeneous resources of a high-performance metacomputer, a parallel application's components (tasks, inter-task communication, i/o) must be carefully scheduled. However, because the resources at hand may be administered by different organizations (although linked by a network) it is difficult to institute a single, system wide scheduler that would be capable of effectively coordinating execution across all domains. Moreover, since the resources are shared, the fraction of maximum performance each can deliver to an application varies over time as a result of contention between competing applications. Resource heterogeneity adds an additional complication since the way in which an application uses a particular resource or resource type dramatically affects the performance that resource car actually deliver to the application.

Application Level Scheduling (AppLeS) [3] is an approach to metacomputer application scheduling in which each application is intergrated with a customized scheduler. Using application-specific performance models and dynamically gathered system information, the AppLeS scheduler attempts to make the best use of the resources that are available to its application at the time the application executes. Each resource is viewed by the scheduler strictly in terms of the potential performance it can deliver. Both the type of the resource and its expected load at the time the application will use it are considered when a schedule is determined. Together, the application and the scheduler for a "resource aware" program which seeks to optimize its own performance using the resources made available to it by the metacomputing system.

In this paper, we describe an application level scheduler designed for a parallel gene sequence library comparison application written for the Mentat [8] prototype metacomputing system. By default, Mentat employs a variant of workstealing [9], to schedule all applications that execute within its environment. We contrast the performance of this system-provided default method with an AppLeS-determined static schedule derived at runtime, and an AppLeS hybrid method based on a combination of runtime scheduling and workstealing. This comparison is important for two reasons. First, the sequential tasks that compose most parallel gene sequencing applications have data-dependent execution profiles. Since the duration of each task varies as a function of its inputs, it is difficult to predict ahead of time. As a result, workstealing type dynamic scheduling techniques are typically employed for such applications. Idle workers "steal" work from busy ones thereby ensuring good resource utilization. We wished to compare the AppLeS approach, which is technique based on application-derived information and dynamically generated resource performance forecasts, to the common approach for a parallel application with hard-to-predict task execution times. The second goal of this work was to investigate a general framework for master/slave computations in which the master process is responsible to distributing work to a set of parallel slaves, either on-demand or proactively. We believe the Mentat implementation we chose (called *complib* which is described more fully in section 3) to use constitutes a reasonable exemplar, in terms of its performance characteristics, for this application class. As such, *complib* was designed assuming that the default Mentat method would yield the best performance in a metacomputing setting.

In Section 2, we describe the problem of comparing biological sequence libraries. In Sec-

tion 3, we describe the implementation structure of the *complib* sequence library comparison application. In Section 4, we describe the AppLeS algorithm we chose for *complib*, and in Section 5, we present the performance of the application in different environments executing with different problem sizes. We conclude in Section 6 with an evaluation of the AppLeS methodology, and a description of future work.

2 Biological Sequence Library Comparison

The primary structual model for many protiens can be represented as a sequence of amino acids. Genetic analysis of DNA, for example, often focuses on the identification base-pair sequences within the molecule itself. Determining these sequences, a process known as sequencing, has become efficient enough that it is commonplace for scientists to sequence as a first step toward determining the structure and function of a protein. Newly discovered sequences can be compared to all known sequences to find others with potentially similar structure, function, and origin.

The desire to compare new sequences against large sequence databases creates a demand for computational resources. The *complib* application implements biological sequence library comparison in a metacomputing environment. Individual sequence comparisons are executed using the FASTA sequential algorithm, described below. Many individual sequence comparisons are executed during library comparison, (described in Section 2.2) and since they are independent, the entire set of comparisons may be done in parallel.

2.1 Sequence Comparison

Sequence comparison is a type of substring matching operation where sequence similarity is scored based on biological criteria. Pairs of sequences that match with a higher score are considered to be more similar, potentially sharing common function and origin. The score of a match increases for each matching element, or identity, and decreases when elements are substituted, inserted, or removed.

The score associated with each substring pair is a function of biological interaction based on mutation and not necessarily minimal structural difference. For example, without using knowledge of the English language, the words *friend* and *fiend* would seem quite similar, and the words *friend* and *friendly* less so. Simple rules based on minimum structural difference would make the comparison of *friend* and *fiend* yield a higher score than *friend* and *friendly*. Using more complicated rules based on the semantics of the words, a better scoring is possible. Such rules exist for comparing proteins. One of these rules is based on a table called a Point Accepted Mutation (PAM) matrix. This matrix describes the the likelihood of substitution of one amino acid by another, and was derived by examination of the small variations between known related sequences. This matrix is then used to determine the scoring penalty for a mismatched element.

FASTA is one of several algorithms used to compare gene and protein sequences. It is composed of four steps. First, groups of identities between sequences are found. Identities

are subsequences of identical elements, and the minimum length of these identities is parameterizable. The ten most promising matching regions of identity are rescored using a PAM matrix. In the third step, these regions may be joined together, so that the best combination of substrings can be evaluated further. The highest scoring joined regions are then rescored using a modification of the more rigorous Smith-Waterman algorithm, restricted to a band around the highest scoring initial region. [16]

Choosing a particular algorithm and tuning its parameters effects the tradeoff between *sensitivity* (finding distantly related sequences), *selectivity* (discriminating unrelated sequences), and *speed*. It is likely that many of the matching pairs of sequences share little function or origin, and attempting to separate these from relevant matches by parameter tuning is generally worthwhile. FASTA is just one algorithm that can be used to compare sequences. Others include the more rigorous Smith-Waterman, [17] and BLAST [1].

Several factors can influence the execution time of this comparison, including the length of the sequences. We examined the distribution of the lengths of the sequences in the National Institute of Health's GenBank **??** database of all known nucleotide and protein sequences. After examining the data, we chose to model the execution profile of an individual sequence comparison task using exponential distribution. We discuss this choice more fully in Section 4.1. Although the time to compare individual sequence pairs is non-deterministic, the time to compare large groups of sequences should be relatively more predictable based on the well-behaved statistical properties of this distribution.

2.2 Library Comparison

Gene sequence library comparison consists of many individual sequence comparisons. Each sequence of amino acids or base pairs in a "source" library is compared to every sequence in the "target" library. We visualize the computation space in Figure 1. Source and target libraries are stored as lists of variable-length sequence structures. Results structures have a statically defined size and contain only the score of the match. Since each sequence comparison is independent, the entire library comparison may be performed in parallel. Complib, the library-comparison implementation we discuss in the next section, exploits this parallelism for performance.

Figure 1: Comparison space. Each sequence in the "source" and "target" arrays must be compared to form a two dimensional array of results structures.

Biologists using this system might submit a set of newly discovered sequences to compare against all known sequences. They would then retrieve information about those similar

sequences in order to hypothesize the function of the new sequences. [4] In this case, the number of sequences in one library is much greater than the number of sequences in another library. This is the type of problem we use when evaluating the performance of the application in Section 5.

It is important to note that the size of these libraries doubles every year. [6] Metacomputing resources can be used to handle problems with data sizes too large to manipulate in the physical memory of any one machine.

3 Complib

Complib is a metacomputer application for comparing biological sequence libraries using the FASTA algorithm, written in Mentat [8]. Mentat is both a run-time system for metacomputer resource management [7] and a programming language based on C++. Parallel tasks in this object-oriented system are contained within Mentat objects, and these Mentat objects are distributed on the different machines in the metacomputer. Communication is accomplished in this object oriented system by passing parameters and return values.

The *complib* application distributes chunks of the source and target libraries to different machines in a metacomputer, so that sequence comparison takes place in parallel. Individual machines compare these chunks of the libraries using a sequential sequence comparison algorithm, in this case FASTA. The distributed resources of the metacomputer allow us to compare more sequences, potentially using more sensitive and computationally demanding comparison algorithms. [10]

3.1 Master-Slave Paradigm

This application is structured using the master-slave paradigm shown in Figure 2. Metacomputers are well suited to applications with this structure since its large computation-to-communication ratio yields performance even with slow networks. A master process distributes work to slave processes that then return the results of the computation to the master.

Work may be placed by the master, as shown by the downward arrows in the work distribution phase of Figure 2, or requested by idle slaves, as shown by the upward arrows. Downward arrows imply *placement*, where a scheduler has knowledge of the slave's performance and can determine an appropriate piece of work to allocate. Upward arrows imply workstealing or *replacement*, where idle slaves request work. Placement generally incurs lower overhead since it is done once just after execution begins, while workstealing can be accomplished with very little scheduling complexity.

3.2 Program Architecture

Mentat is a programming language with an object-oriented approach to parallelism, so *complib* is organized as a collection of objects that may execute in parallel. There are three

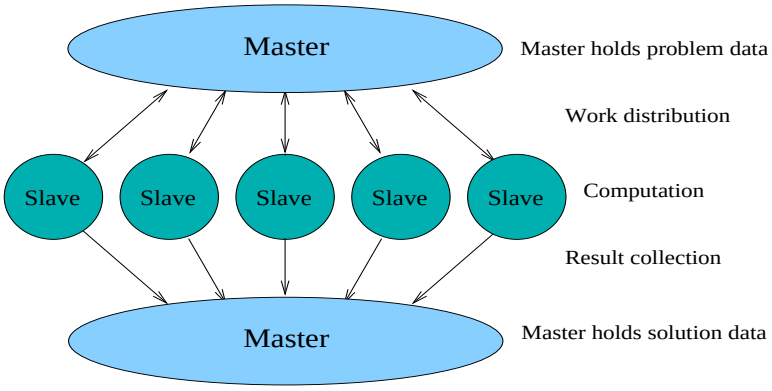

Figure 2: Master-slave communication and structure template.

classes of object: libraries, workers, and collectors, shown in Figure 3. Genome library objects load sequence libraries from disk, then disseminate chunks of these libraries to the worker objects. There are two library objects during execution: one for the source library, and another for the target.

In this object-oriented system, computation takes place as the result of method invocation. In this case, the worker objects export a compare() function that accepts library chunks as parameters and returns a two dimensional array of results structures. These results structures consist of the scores generated by the sequence comparison algorithm, and are forwarded to the collector object. These results are then sorted by score by the collector object.

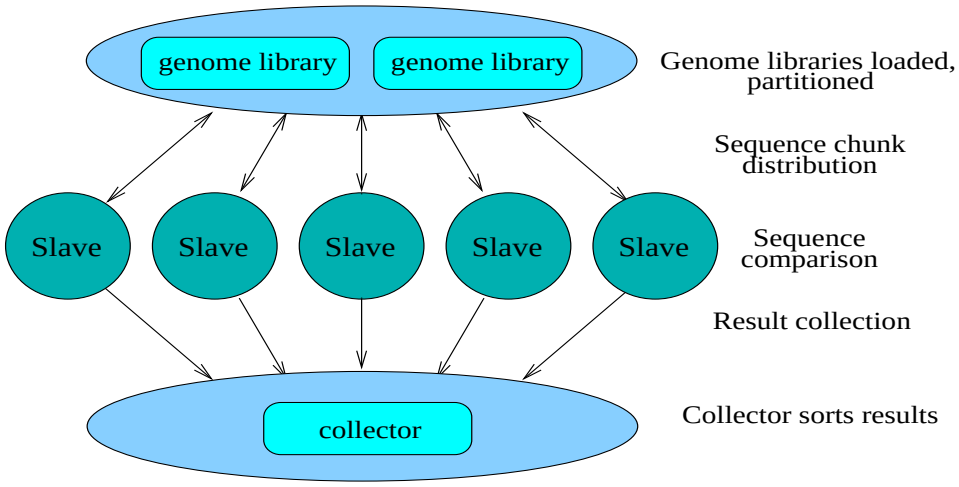

Figure 3: Complib data flow.

Although there are two genome library objects, only the target genome library handles the startup of worker objects and the allocation of sequences. Logically, these two library objects represent the master process for work distribution, and the collector object represents

the master for the purpose of result collection.

Worker objects are a type of *regular* Mentat object, which allows the system to determine where and how many objects will be started. Regular objects hold no state between method invocations, and since each comparison takes place independently, this is appropriate. *Persistent* Mentat objects can reside on a specified host, and are created one at a time at the application level. Library and collector objects are both persistent, since they both store state between method invocations. The collector object stores a partially assembled array of results between functions, and the library objects store the genome library to be distributed.

3.3 Mentat System Scheduling Policy

Currently, *complib* relies on the Mentat scheduler.[9] Chunks of a parameterizable maximum size are placed on a system-managed queue. This block-style data distribution is shown in Figure 4. Effectively, when objects run out of work, they request a pair of chunks, one from the source library and another from the target from this queue. In the actual implementation of the Mentat scheduler, congested nodes search for idle nodes, in order to promote load sharing. Since worker objects are members of a *regular* Mentat class, the Mentat scheduler will start as many worker objects as necessary, and chooses where they will be started. The Mentat scheduler also chooses where the library and collector objects will reside, since the application leaves this unspecified.

Startup overhead and load imbalance overhead reduce application performance. Adjusting the size of the chunks that are placed on this queue trades startup overhead for load imbalance. By increasing the chunk size, the cost of startup overhead is decreased, since there are fewer chunks on which to operate. But a large chunk size reduces potential parallelism, as it becomes likely that one slow processor will operate on its last partition while all other processors are inactive. A small chunk size increases the potential parallelism, but also increases the communication overhead, since additional messages must be exchanged.

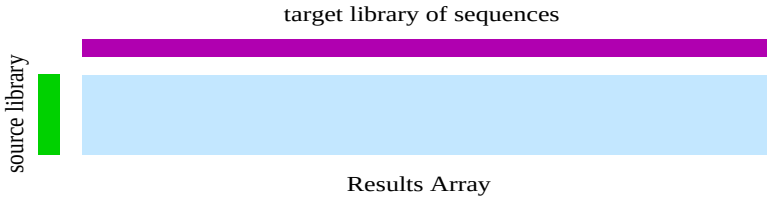

Figure 4: Default block distribution of data. Block size is configurable and affects application runtime.

Mentat scheduling of *persistent* objects, like the libraries and collector, is essentially round robin. This works well for homogeneous clusters of workstations on a single network subnet. More sophisticated scheduling techniques can be used to improve performance in a heterogeneous environment. In order to reduce these overheads, we would like to place both the library and collector objects on hosts with good communication performance to the worker objects. We would also like to partition the data so that each processor receives one

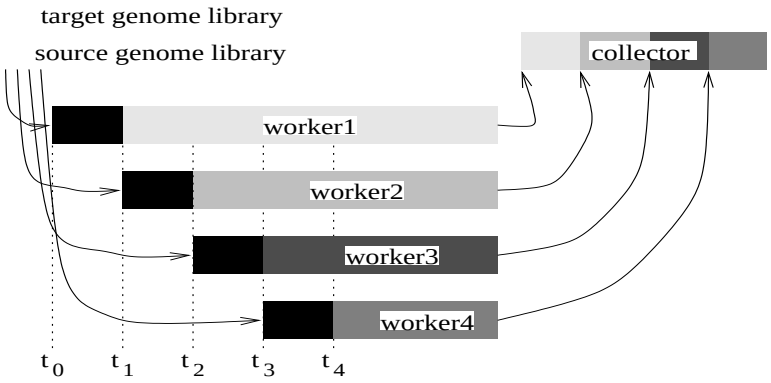

Figure 5: Basic complib application model. Black areas show startup overhead. Arrows indicate data flow.

large chunk, then finishes at approximately the same time. This demands an application-specific scheduler since it is the application's particular needs that determine how it should be scheduled.

4 AppLeS

Application Level Scheduling (AppLeS) combines dynamic system performance information with application specific models and user specified parameters in order to produce better schedules. [3] "Better" is defined by the user, but in this case will be taken to mean decreased run time. In this section, we will describe the application model used, and how the scheduler can improve application performance during each phase of execution. We also present a framework for scheduling master-slave applications in a shared, heterogeneous environment.

4.1 Complib Application Model

The *complib* application model can be split into three overlapping phases. First, the workers are started and given gene sequence library chunks to compare. Second, the workers compare each sequence in the "source" chunk to each sequence in the "target" chunk. Finally, the workers return a results structure to the collector. These activities are shown in Figure 5.

Each of the worker objects is a Mentat persistent object. We use persistent objects because in Mentat,[1] persistent objects are the only way to disable the default workstealing policy and allow application-controlled placement. Persistent mentat objects, however, can only be started one at a time. This creates significant startup overhead which the AppLeS must consider. Note that since regular objects are not application schedulable, the AppLeS approach must be able to improve execution enough to recover the additional overhead (imposed by Mentat) associated with using persistent objects.

[1] Mentat is a forerunner of Legion and this problem has been fixed.

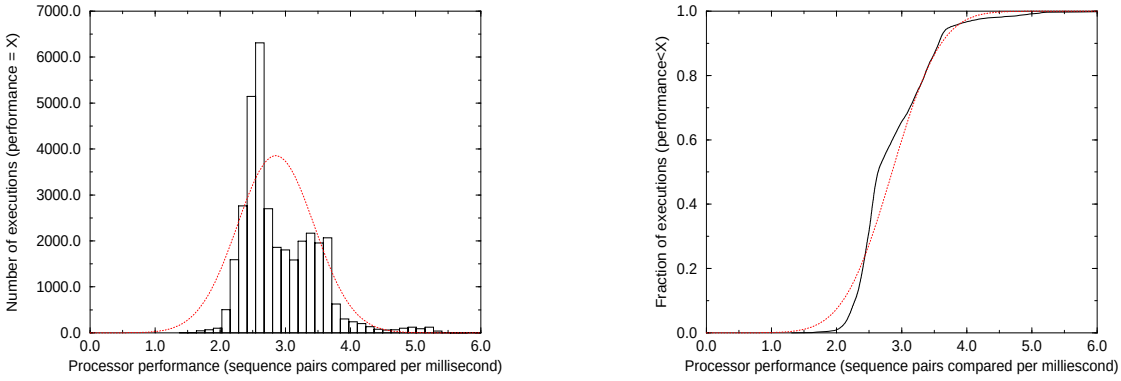

Figure 6: Normal distribution of execution performance. Shown on the left is a histogram of average execution performance for one of the Sun enterprise servers, measured in sequence pairs compared per millisecond. Superimposed on the histogram is the bell curve of a normal distribution. On the right is the same information shown as a cumulative distribution function.

The startup behavior of persistent objects is shown in Figure 5. At time t_0, worker1 is started. The black area is a combination of the time to load the worker's binary code stored on an NFS mounted file system, and the time to receive genome library chunks. As soon as worker1 begins actual execution, at time t_1, worker2 is started. This pattern continues through t_4, when all the workers are executing.

The time to compare source and library chunks is modeled using two parameters:

- a benchmark execution time of a worker task on each possible host in dedicated mode, and
- a predicted measure of percentage CPU availability generated by the Network Weather Service [20].

By dividing the expected execution time of a worker task on a dedicated host by the expected percentage of the time the taks will be able to occupy the host's CPU, we can obtain an expected execution time for the task that accounts for CPU contention. We obtain the benchmark information by using operating system provided CPU-time accounting information. This provides an estimate of the dedicated time required to compare a pair of sequences on each host. In Figure 6, we show a histogram and cumulative distribution of execution times for the FASTA implementation within *complib* over a sample from the GenBank data. As shown in Figure 6, While the histogram does not appear to be modeled well by a normal distribution, note that the cumulative distribution tracks the normal well above the 70th quantile. The scheduling algorithm we describe in Section **??** relies on the ability to determine a *dependability threshold* on execution time prediction. Based on the relationship between the observed data and the normal quantiles, we chose to use an exponential

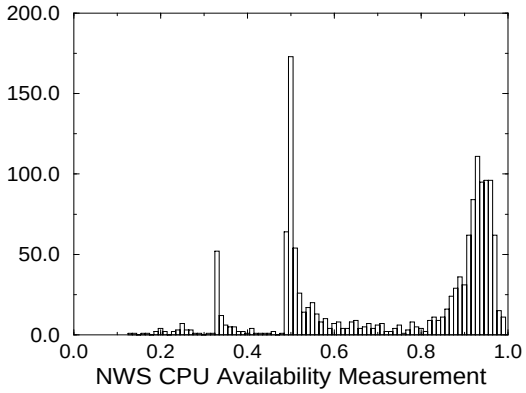

Figure 7: Histogram of CPU availability values provided by the Network Weather Service. This distribution is multimodal.

distribution to model the dedicated execution time of FASTA. Expected CPU availability, however, is not normally distributed [12] as shown in Figure 7.

To predict the percentage of available CPU occupancy that will be possible for each worker task, we use the Network Weather Service (NWS) [20]. The NWS is a distributed performance monitoring and forecasting facility designed to provide schedulers with predictions of the deliverable performance of resources to applications. Currently, the NWS provides CPU availability forecasts, and available network bandwidth and latency predictions to the scheduler, although the NWS can supply other information. [20, 21] For this application, we are most concerned with the fraction of the CPU that the NWS predicts will be available at the time the application will execute, although network performance predictions are also used.

The following formula is used to model the execution time of a worker process in the implementation of *complib*:

$$T_{compare} = \frac{\text{Source chunk size * Target chunk size}}{\text{Benchmark sequence pairs per millisecond}} * \frac{1}{\text{NWS CPU Availability}}$$

After comparison, workers pass a result structure to the collector object. These size of a result structure scales in proportion to the product of source chunk size and target chunk size, and can reach 10 megabytes. Transfer time for result structures of this size is noticeable, and takes place sequentially.

We are concerned primarily with scheduling the computation phases of this application. Although further performance improvements may be gained by scheduling the I/O and collection phases of the program, we chose to focus on the computational phase first as it constitutes the largest fraction of overall execution time. Moreover, substantial structural changes to the existing *complib* application would be necessary to allow such scheduling. In this work, we wished to investigate the effect of AppLeS scheduling techniques in comparison with the existing Mentat scheduling method (which was conjectured to be the "best"

methodology for the application as it was written). As such, we did not wish to change the structure of the implementation in a way which might favor AppLeS.

4.2 Startup Phase & Resource Selection

Given the behavior of persistent objects in Mentat, it is necessary to limit the number of processors we use, and choose the best processors available. Not all resources are useful because of the large startup delay. Note, however, that AppLeS can consider the delay associated with different object-to-host mappings, and choose only those resources (and the time at which they should be used) so that the execution time of the program is optimized. A truly optimal schedule is not feasible as the problem is NP-complete. Instead, the AppLeS employs a time-balancing heuristic [3] which attempt to cause all workers tasks to finish simultaneously. In this work, we extend time-balancing to include the notion of start-up delay so that the AppLeS scheduler may consider the start-up cost associated with Mentat persistent objects.

4.2.1 Library and Collector Placement

Library placement is the process of determining where the library objects will reside. These hosts should be *close* to both data storage and the processors where workers will be run. The term "close" in this context refers to the relative bandwidth performance of the network linking two computational sites. Two hosts are close if the relative bandwidth linking them is high. They are far otherwise. Potential library locations are scored based on the sum of the products of processor power and bandwidth to each host. The host with the highest score is allocated the target library, and the second highest score is allocated the source library and collector object.

$$\text{Library Placement Score} \quad = \quad \Sigma(P_{processor} * Bandwidth_{toprocessor})$$

4.2.2 Worker Placement

The sequential startup represented an important constraint. Since it generally takes between one and two seconds to start a worker object, using 30 processors would take at least 30 seconds. However, using regular Mentat objects (which cannot be scheduled from the application level) the existing implementation could achieve runtimes that were much less than 30 seconds. Clearly, the AppLeS implementation could not use all of the available hosts if it were to be able to amortize the cost of persistent objects.

To select an appropriate subset of resources to use, the scheduler starts by sorting the available hosts by processor power.

$$P_{processor} \quad = \quad \frac{\text{NWS CPU Availability}}{\text{Benchmark sequence pairs per millisecond}}$$

Now that the processors are sorted by power, the most powerful processor will be started first. This maximizes our use of the best processor resources that are available.

The next step is to determine the maximum number of processors that can be used to decrease the overall runtime. Startup cost is modeled as a static 1.5 seconds plus a communication cost, which serves as a reasonable approximation at this stage. This communication portion of the startup cost is calculated using the following formula:

$$TransferTime = \frac{PartitionSize * AverageSequenceLength}{BandwidthPrediction} + 3 * LatencyPrediction$$

The latency prediction is multiplied by three, since that is an estimate of the number of message transfers between libraries and worker at startup.[2] $PartitionSize*AverageSequenceLength$ represents the size of the partition being transferred. Since the lengths of sequences in a chunk are not known until the chunk is assembled, a representative measure of sequence length must be used, in this case, we chose an average. The *AverageSequenceLength* from the GenBank database described in Section 2 was 243 bytes. A similar formula is used to estimate the communication cost involved in transferring the results structure back to the collector, which is not as important at this phase. The scheduler chooses the number of processors that yields the lowest expected execution time by considering successively larger subsets of the sorted processor set in sorted order.

An enhancement to this algorithm would adjust the sorting order using communication performance and an estimate of the data size so that the lower overall utility to the application of distant but powerful processors is considered. We are considering this improvement and others as part of our on-going research.

4.3 Completion Phase & Collector Concurrency

In Figure 5, the collector object executes after all of the workers have finished. This data transfer represents a significant sequential bottleneck. In order to alleviate this problem, the large chunk sent to the workers is split into two smaller chunks. This implementational change allows the collector and the workers to overlap execution.

We show the effect of this placement in Figure 8. At time t_a, worker1 finishes its first partition, and forwards the results structure to the collector. It begins execution on the second partition immediately, since the library chunks have already been forwarded through the mentat system to that host. The collector receives the results, and waits until the next structure arrives at t_b.

We believe that additional performance is possible by splitting the large data allocation into more than two chunks. However, more substantial modifications to the structure of *complib* seemed to provide AppLeS with an unfair advantage. By providing this level of overlap, we hoped to ensure a fair comparison between the two methodologies.

[2]Due to the object-oriented macro data flow model, the number of actual message transactions is hidden from the application.

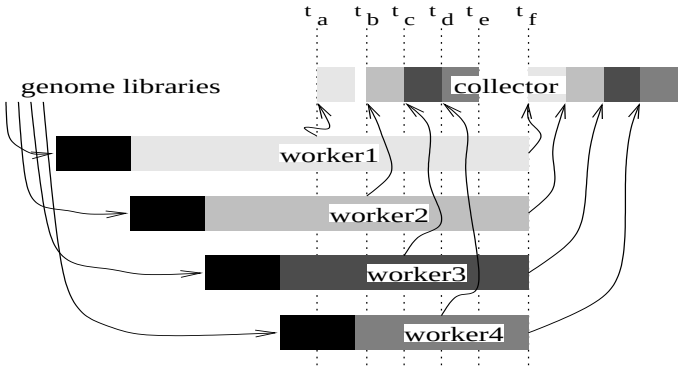

Figure 8: Placement of two half-size partitions to reduce collector bottleneck.

4.4 Comparison Phase & Hybrid Vigor

The performance improvements gained using a static placement strategy (determined at runtime just before execution begins) depend on the quality of the information used. Incorrect benchmarks or poor performance predictions from the Network Weather Service can cause the scheduler to make the wrong decisions. These wrong decisions result in costly load imbalance. There are many factors, both operating system and algorithm dependent, that influence the time it will take to compare sequences. In the general case, even the most refined benchmarks and best predictions of available performance will have an associated uncertainty, and this uncertainty equates to a variance in execution time of the worker processes.

The goal of the scheduler, then is to avoid load imbalance while still enjoying the advantages of static placement. To do so, the scheduling model can be separated into two phases: *placement* and *replacement*. *Placement* is the process of allocating work to processors once, before execution begins, using the best performance estimates that are available. *Replacement* is the allocation of the remaining work using a dynamic technique, in this case, Guided Self-Scheduling. [15] This strategy is shown in Figure 9 for comparison with Figures 5 and 8.

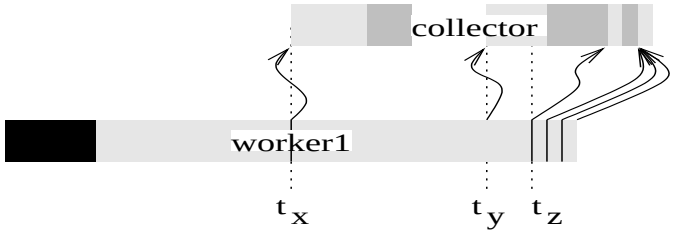

Figure 9: Single worker object's partitions. At time t_x, work on the second half-partition begins. Time t_y is the boundary between placement and replacement, as the worker begins using some dynamic technique such as GSS or workstealing. At time t_z, the worker executes a second dynamically determined partition.

It is the scheduler's job to choose the *demarkation point* between static placement and

dynamic allocation so that the load imbalance that might result from a bad placement is mitigated by the dynamic technique. Simultaneously, the scheduler must try and make the statically placed tasks as large as possible to reduce execution time overhead associated with dynamic scheduling. Note that this scheduling model is general enough to include most master/slave implementations in which the master can dole out the work on demand. To determine the demarkation point the scheduler must consider the uncertainty associated with each execution time estimate. If the estimates are relatively accurate, the scheduler can profitably assign more work using static placement and less work using a dynamic technique. If the uncertainties are high, the reverse is true.

The algorithms used for placement and replacement, however, are not specified by the model. Much work, for example, has been done to improve dynamic scheduling techniques [14, 2, 15, 13]. The scheduler need only be able to consider the non-amortizable overhead cost of each in order to determine is benefit relative to a given placement algorithm. Similarly, a variety of static placement algorithms such as time-balancing or recursive bisection may be employed. In this work, we chose time-balancing and Guided Self-Scheduling to instantiate the model as both have demonstrated their ability to achieve high-performance levels in a variety of settings. We combine the time-balancing algorithm for placement, with a weighted adaptation of Guided Self-Scheduling (GSS) to allocate the remaining computation. GSS allocates successively smaller chunks of the computation in order to avoid overhead early on, while still providing parameterizably even finishing time.

4.4.1 Boundary Placement

In order to take the best advantage of the low overhead of placement, the boundary should be chosen so that placement is used for as much of the data as possible. We estimate how much of the work can be placed by estimating the variance in execution time as a function of the variance in CPU availability. Execution time also varies as a function of the sequences being compared. The distribution of CPU time required per sequence compared (averaged over chunks of at least 1000 sequence comparisons) is shown in Figure 6. The boundary is chosen on a per-object basis so that a maximal portion of the work that can dependably be completed is placed. After an object's dependable portion of the work is complete, it requests additonal work as part of the replacement phase.

Since the distribution of CPU availability values is often multimodal, (see Figure 7) we do not assume normality, but rather use an estimate of variance provided by the NWS prediction modules. [19] The NWS tracks the error associated with each of the predictions of resource performance it supplies. We incorporate that information into the calculation of the placement/replacement boundary by using a multiple of the mean squared prediction error as a confidence interval about the prediction. That is, the scheduler assumes that the performance that the application will eventually receive will fall reliably within the interval provided by the NWS. It can therefore count on receiving at least as much resource performance as indicated by the leading edge of the interval.

$$
\begin{aligned}
ExpectedPerformance &= \text{NWS CPU Availability} * \text{Benchmark} \\
DependableCPUAvailability &= \text{NWS CPU Availability} - \\
&\quad 3 * \text{NWS CPU Availability Mean Percentage Error} \\
DependablePerformance &= DependableCPUAvailability * \text{Benchmark}
\end{aligned}
$$

The number of sequences proportional to the fraction

$$\frac{ExpectedPerformance - DependablePerformance}{ExpectedPerformance}$$

is held in reserve for replacement. This calculation takes place for each machine, so that machines with high load variance are allocated a smaller portion of the problem and contribute more to the replacement work pool.

Sequences allocated through placement are shown in Figures 10 and 11. In Figure 10, the difference between *expected* and *dependable* performance is shown. Regions shaded in black represent the difference between these, to be held in reserve. In actual implementation, sequences are allocated contiguously, and the reserved work actually represents the last set of sequences, as shown in Figure 11.

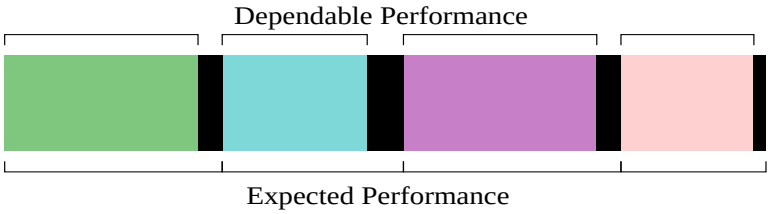

Figure 10: Conceptual view of expected and dependable performance. Sequences are placed based on the dependable performance of a processor. Sequences to be compared are held in reserve based on the difference between expected and deliverable performance.

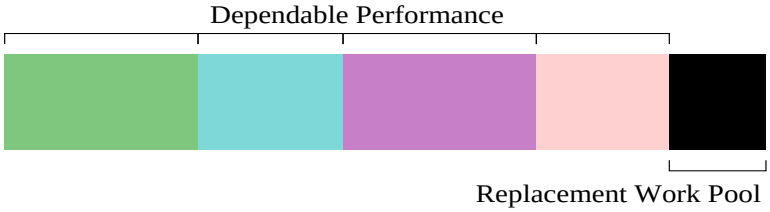

Figure 11: Implementation view of expected and dependable performance. Sequences are allocated in contiguous blocks. Work kept in reserve can be allocated to any worker object that requests more work, and is not tied to any individual machine.

5 Results

In this section, we compare the execution time of the original Mentat implementation of *complib* with that of an AppLeS-enhanced version. In all experiments, we execute both versions back-to-back multiple times on non-dedicated machines and networks, and report the range execution times along with the average. Since the overheads are not invariant with respect to problem size, we compare executions using three different representative sequence libraries: one small, one meedium, and one large. Similarly, we wish to reflect different possible metacomputing settings, so we use clusters of resources representing small, medium, and large levels of geographic dispersion, the largest of which employs resources located on opposite coasts of the United States.

Since application run-times fo the original *compilb* depend upon the size of the library chunk-size, we attempted to the best chunk size for each cluster empirically. That is, we conducted several hundred runs of the Mentat *compilb* on each cluster, using different blocking factors. In all of the experiments shown below, we report the performance of the original Mentat *compilb* using the chunk-size we observed to perform best on each cluster size. In contrast, the AppLeS-enhanced version chooses its own partitioning automatically based on performance forecasts.

Application run-times are also determined by various other parameters, like where the objects are placed by the system, and how loaded the resources are. Consequently, one value is not enough to describe the performance of these schedulers. We present the mean execution time, along with a range of run-times seen over the course of 30 runs. These 30 executions of AppLeS-scheduled and Mentat-scheduled *complib* were repeated at three different data sizes on three different platforms: small, medium, and large, shown in Figure 12. Effectively, these represent local area, "enterprise"-scale, and national-scale platforms. The three data sizes are labeled small (20x10,000 sequences), medium (20x32,000), and large (20x120,000).

5.1 Heterogeneous Platforms

We chose three different platforms so that scheduling methodologies could be compared in different environments. The small cluster is intended to show machines on a local area network, the medium adds machines in another building reflecting metacomputing at an organiational or enterprise-scale, and the large adds a machine across the country. All resources in each cluster, during each run were being operating in a non-dedicated production computing mode. Each *Complib* execution had to compete with potentially contending applications and had to use non-dedicated, shared networks.

The small cluster consists of five machines, all located at the San Diego Supercomputer Center. These include two four processor Sun Enterprise servers, two two-processor Sun Ultra workstations, and one Sun Sparc-4 workstation, to total eleven processors. The network configuration is also heterogeneous: the Enterprise servers are connected to each other by fast (100 megabit) ethernet, other machines are connected by 10 megabit ethernet, and the processors within each multiprocessor machine communicate via shared memory.

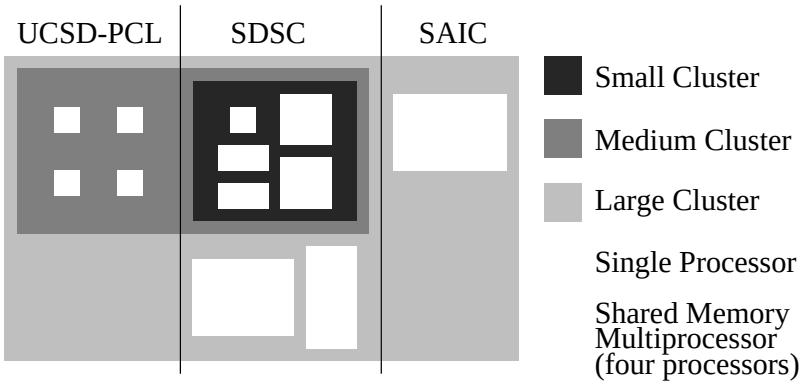

Figure 12: Cluster sizes chosen. Workstations and servers from UCSD's Parallel Computation Lab (PCL), the San Diego Supercomputer Center (SDSC), and Science Applications International Corporation (SAIC) in Virginia were used.

The medium cluster adds four single processor machines from the Parallel Computation Lab (PCL) at UCSD. There are two Sun Ultra workstations, one Sun Sparc-5 and one Sun Sparc-10. These four machines are connected by a slow ethernet serving the rest of the PCL resources, and over the shared campus backbone to SDSC. The large cluster adds a 12 processor Sun CS-6400 at SAIC in Virginia, connected via a national ATM internet, and two additional multiprocessor Sun Enterprise servers at SDSC.

We also compare two different AppLeS scheduling techniques to the Mentat-scheduled version of *complib* in each setting, for each problem size. The first uses an AppLeS to determine a static *placement* only of the computations. The placement is determined at run-time just before execution begins, all sequence comparisons are assigned, and no *replacement* phase is executed. The second AppLeS uses a hybrid combination of static, run-time *placement* and dynamic *replacement* based on Guided Self-Scheduling. The bounadry between these two phases is determined by the prediction accuracy reported for each resource by the NWS as a quantitative measure of that prediction's quality. Use of such *Quality of Information* or *QoIn* metrics is the subject of other, on-going research efforts within the AppLeS research group at UCSD [**?**].

5.2 Complib Execution Performance

The largest performance improvement provided by application level scheduling of *complib* was seen on the medium problem size on the medium sized cluster (middle graph of Figure 13). AppLeS run-time static scheduling ran an average of 59.7the hybrid approach increased that performance gain to 69

The largest difference between the two AppLeS approaches (AppLeS run-time static and the hybrid combination of *placement* and *replacement* strategies) seen at the small cluster size and large data size (Figure 13, top graph). The hybrid was 34.4than the run-time static. The reason for this performance difference is that that the accuracy of the run-time static

prediction degrades over time. A prediction made at the beginning of a short execution run is more accurate for the duration of the run, than for a long execution run. The hybrid approach considers the effect of this inaccuracy and compensates for it by using dynamic Guided Self-Scheduling when prediction inaccuracy would cause a load imbalance.

Our hybrid approach provided the worst performance on the small cluster with the medium data size (Figure 13, top graph). The hybrid actually ran 1.1static partitioning alone. It is likely that this is a result of very little contention on the machines at the time these were executed: workstealing simply was not necessary to provide balanced load. We had hoped that the AppLeS would automatically recognize this condition and use only static *placement* in response. Since the performance penalty was small, however, we did not attempt to tune the AppLeS further to eliminate this discrepancy.

In general, however, the AppLeS improvement over the original Mentat scheduling method increases with both problem size and cluster size. Similarly, the improvement of hybrid *placement* and *replacement* over run-time static AppLeS scales with problem and cluster size. As metacomputing becomes more prevalent, we believe that these AppLeS based techniques will offer even greater performance advantages.

6 Conclusion

The goal of this work was to investigate the comparison of application-level scheduling techniques with a commonly used dynamic scheduling method (dynamic workstealing) using an application that was designed and tuned to work with this common method. In addition, we wished to define a general framework for scheduling master-slave parallel applications with data dependent execution profiles in a shared, heterogeneous environment.

To do so, we chose a genetic sequencing application based on the FASTA algorithm implemented for Mentat called *complib*. The implementation of *complib* was specifically designed to work with the native Mentat workstealing scheduler. We attempted to fit this implementation with an AppLeS (application-level scheduler) without rewriting it to favor and the implementation of the the AppLeS. That is, we wished to compare the best possible Mentat implementation with a simple application of the AppLeS techniques, favoring the Mentat implementation whenever possible.

In all our the experiments we conducted, the AppLeS-enhanced version of *complib* outperformed the original Mentat version in production metacomputing settings by as much as 69relative performance improvements of the AppLeS-enhanced versions scale with problem size and the size of the metacomputing resource pool leading us to believe that AppLeS will become an even more effective approach as metacomputing matures.

The most effective technique we have demonstrated combines static, run-time determined *placement* of some fraction of the work with dynamic *replacement* of the uncomputed fraction to ensure load-balance. Controlling the balance between *placement* and *replacement* requires the AppLeS scheduler to evaluate the *Quality of Information* associated with each prediction. The function of the AppLeS itself does not depend on a specific *placement* or *replacement* algorithm. In the experiments, we used time-balancing and Guided Self-Scheduling, but

other static and dynamic techniques could be used instead. As such, we believe that this technique represents a general framework for master/slave parallel metacomputing computations in which the master distributes uncomputed work to the slaves. As part of our future research efforts, we plan to investigate different applications from this application class as well as different static *placement* and dynamic *replacement* algorithms within the context of this framework.

References

[1] S. Altschul, W. Gish, W. Miller, E. Myers, and D. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215:403–10, 1990.

[2] I. Banicescu and S. F. Hummel. Balancing processor loads and exploiting data locality in irregular computation. Technical Report RC 19934, IBM, 1995.

[3] F. Berman, R. Wolski, S. Figueira, J. Schopf, and G. Shao. Application level scheduling on distributed heterogeneous networks. In *Proceedings of Supercomputing 1996*, 1996.

[4] M. Bishop, editor. *Guide to Human Genome Computing.* Academic Press, 1994.

[5] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *International Journal of Supercomputer Applications*, 1997.

[6] E. Green, editor. *Genome Analysis: A Laboratory Manual.* Cold Spring Harbor Laboratory Press, 1997.

[7] A. Grimshaw. The mentat run-time system: Support for medium grain parallel computation. In *Proceedings of the Fifth Distributed Memory Computing Conference*, pages 1064–1073, April 1990.

[8] A. Grimshaw. Easy-to-use object-oriented parallel programming with mentat. *IEEE Computer*, May 1993.

[9] A. Grimshaw and V. Vivas. Falcon: A distributed scheduler for mimd architectures. In *Proceedings of the Symposium on Experiences with Distributed and Multiprocessor Systems*, pages 149–163, March 1991.

[10] A. Grimshaw, E. West, and W. R. Pearson. No pain and gain! – experiences with mentat on a biological application. *Concurrency: Practice and Experience*, 5(4):309–328, June 1993.

[11] A. S. Grimshaw, W. A. Wulf, J. C. French, A. C. Weaver, and P. F. Reynolds. Legion: The next logical step towrd a nationwide virtual computer. Technical Report CS-94-21, University of Virginia, 1994.

[12] M. Harchol-Balter and A. Downey. Exploiting process lifetime distributions for dynamic load balancing. In *Proceedings of the 1996 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, 1996.

[13] S. Hummel, J. Schmidt, R. Uma, and J. Wein. Load-sharing in heterogeneous systems via weighted factoring. In *Proc. 8th ACM Symp. on Parallel Algorithms and Architectures*, pages 318–328, 1996.

[14] S. Lucco. *Adaptive Parallel Programs.* PhD thesis, University of California, Berkeley, August 1994.

[15] C. Polychronopoulos and D. Kuck. A practical scheduling scheme for parallel computers. *IEEE Transactions on Computers*, C-36(12):1425–1439, December 1987.

[16] W. R.Pearson and D. J. Lipman. Improved tools for biological sequence comparison. *Proc. Natl. Acad. Sci*, 85:2444–2448, April 1988.

[17] T. Smith and M. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147:195–197, 1981.

[18] T. Tannenbaum and M. Litzkow. The condor distributed processing system. *Dr. Dobbs Journal*, February 1995.

[19] R. Wolski. Dynamically forecasting network performance to support dynamic scheduling using the network weather service. In *Proc. 6th IEEE Symp. on High Performance Distributed Computing*, August 1997.

[20] R. Wolski. Dynamically forecasting network performance using the network weather service. *Cluster Computing*, 1998. available from http://www.cs.ucsd.edu/users/rich/publications.html.

[21] R. Wolski, N. Spring, and C. Peterson. Implementing a performance forecasting system for metacomputing: The network weather service. In *Proceedings of Supercomputing 1997*, November 1997.

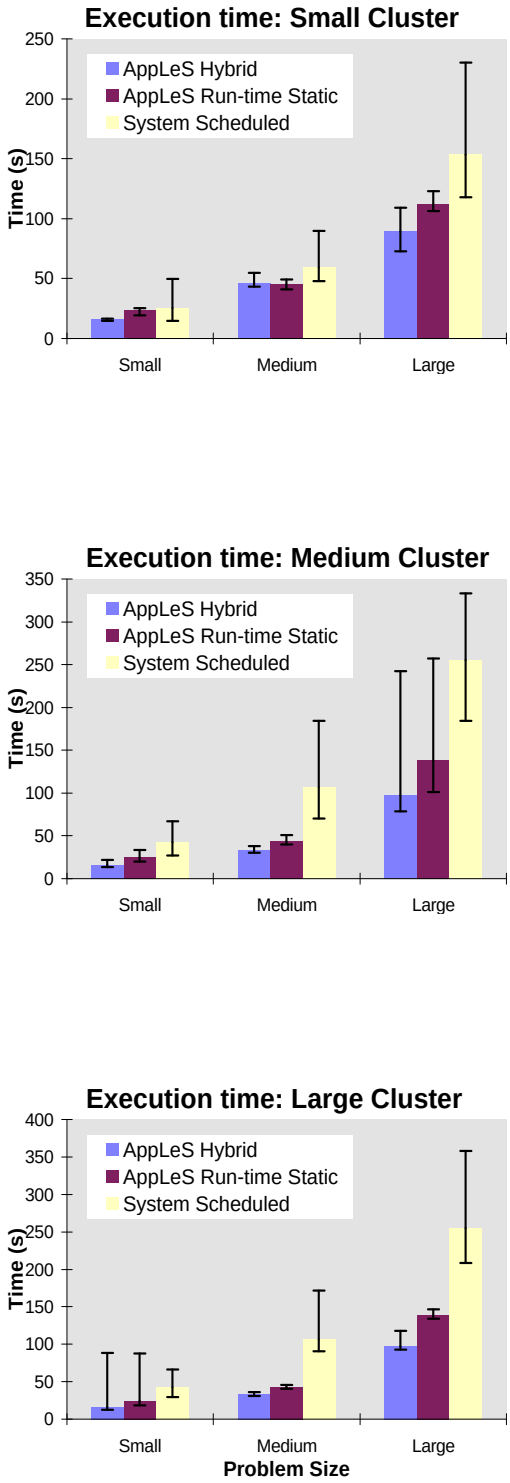

Figure 13: Execution times for small cluster (top), medium cluster (center), and large cluster (bottom). Bar height is equal to the average of 30 execution times. Error bars cover the range of all 30 execution times.