Gene Sequence Alignment on a Public Computing Platform

Stephen Pellicer, Nova Ahmed, and Yi Pan
Department of Computer Science,
Georgia State University, Atlanta, GA, USA
spellicer1@student.gsu.edu, nahmed2@studnet.gsu.edu, pan@cs.gsu.edu

Yao Zheng
College of Computer Science,
Zhejiang University, Hangzhou, Zhejiang, China
yao.zheng@zju.edu.cn

Abstract

Public computing can potentially supply not only computational power but also memory and short term storage resources to grid and cluster scale problems. Gene sequence alignment is a fundamental computational challenge in bioinformatics with attributes such as moderate computational requirements, extensive memory requirements, and highly interdependent tasks. This study examines the performance of calculating the alignment for two 100,000 base sequences on a public computing platform utilizing the BOINC framework. When compared to the theoretical, optimal sequential implementation, the parallel implementation achieves speedup by a factor of 1.4 and at the point of maximum parallelism and ends with a speedup of 1.2. This speedup factor is based on extrapolation of the sequential performance of a segment of the problem. This extrapolation would require a theoretical sequential machine with approximately 37.3 GB of working memory or suffer performance degradation from use of secondary storage during the calculation.

1. Introduction

Bioinformatics has emerged as a popular and fruitful field of study drawing from many scientific disciplines including computer science, biology, chemistry, and physics. The computational challenges faced by bioinformatics feature enormous data sets, intense computation, and a wide variety of unanswered questions. Gene sequence alignment is one such problem that serves as an initial step in many of the problems in bioinformatics. This problem is often addressed using dynamic programming algorithms [1].

Public computing, or Internet computing, is an architecture designed to harness the idle cycles of Internet connected workstations. The Berkeley Open Infrastructure for Network Computing (BOINC) is a general framework designed to implement this new architectural paradigm. The system software is being developed at the Space Sciences Laboratory at the University of California, Berkeley with project leader Dr. David P. Anderson [3]. This system is currently deployed for the SETI@home project and a growing number of additional projects. The system is designed as a software platform utilizing computing resources from volunteer computers.

While the use of public computing has seen traction in large scale projects such as SETI@home, research into use of the platform for smaller, Grid and cluster scale projects has seen little attention. Rosenberg looks at public computing scheduling in the context of fault tolerance and efficient scheduling with highly interdependent tasks [4]. Grid related research examine many issues shared with public computing such as the difficulty of data co-location [5]. Additionally, systems such as Condor utilize resource scavenging techniques similar to public computing, and there is work to integrate Condor with Grid systems [6].

The gene sequence alignment problem was chosen primarily for the interesting structure of task dependencies. In addition to highly interdependent tasks, the calculation requires a large amount of memory to store and calculate the alignment matrix. The other attributes of the problem, data communication costs and computational intensity, are relatively light compared to the task dependency. This experiment explores the issues with scheduling dependent tasks of a computation on the public computer architecture.

Proceedings of the 2005 International Conference on Parallel Processing Workshops (ICPPW'05)
1530-2016/05 $20.00 © 2005 IEEE

2. Problem Description

The goal of this experiment is to align two gene sequences according to a scoring system penalizing gaps and mismatches between the sequences. An example takes two sequences of genes composed of the bases A, C, G, and T:

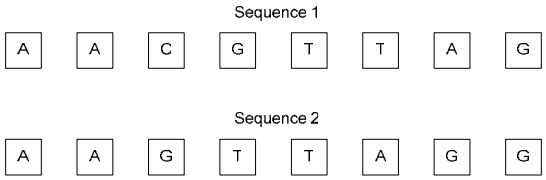

Figure 1 Sequence Example

These sequences can be aligned in a number of ways taking into account inserting gaps and accounting for mismatched bases. A scoring system can penalize the addition of gaps and mismatches. Different alignments will produce different scores. Take, for example, a gap penalty of -2, a mismatch penalty of -1, and a match score of 1. The original alignment shown gives the following score:

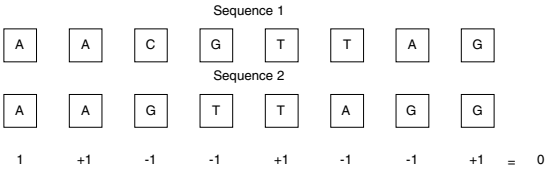

Figure 2 Scoring Example

Through the addition of gaps, the score can be improved despite gap penalties:

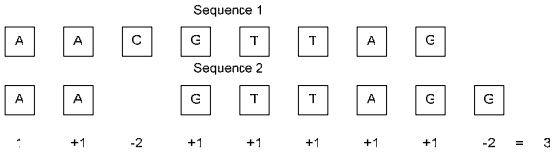

Figure 3 Improved Score

All possible scoring combinations are on the order of $O(mn)$ when aligning an *m* length sequence with an *n* length sequence. The experiment examined in this paper calculates and finds the best alignment. The scale of this problem is $O(mn)$ for both the number of comparisons required and the memory used.

3. BOINC Architecture

The BOINC architecture consists of a server complex handling scheduling, central result processing, and participant account management, a core software client running on participant nodes, and project specific client software running on participant nodes (Figure 4). The server complex consists of a database, web server, and five BOINC specific processes. These server components are designed to run on either a single system or splitting the various functions across several servers. Communication between participant nodes and the server complex is handled via standard HTTP.

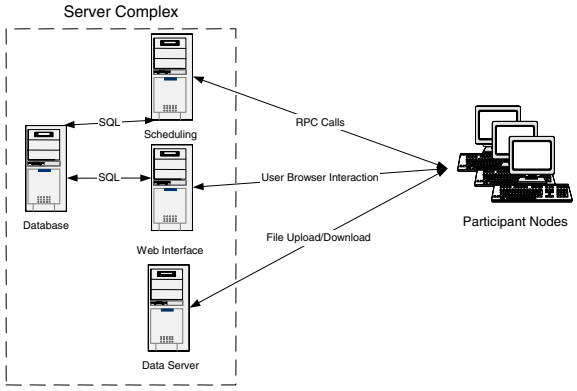

Figure 4 Basic BOINC Architecture

Computing resources are allocated to the computational problem by the assignment of workunits. Workunits are application specific and represent a subset of the entire computational problem. Where workunits are the basic input mechanism for participating nodes, result units are the basic output mechanism.

3.1. Default BOINC Scheduling

The default scheduling mechanism in the BOINC architecture pays very little attention to optimizing performance or throughput. Because BOINC is designed for very large scale participation for enormous computations, detailed implementation of scheduling is, for the most part, unnecessary. The study of computational problems such as this gene sequence alignment calculation is to examine the viability of the BOINC architecture for problems of this scale.

The design of the BOINC scheduling algorithm pays close attention to the communication overhead imposed by a restricted interconnection network. In a large deployment, slow and unreliable network links can be greatly impacted by the polling nature of the BOINC client software communication with the scheduling server. Communication can bottleneck due to three major factors:

- Large amounts of workunit data
- Large amounts of result data

Table 1 Participant Nodes

Processor Configuration	Operating System	Number
Intel® Pentium® 4 CPU 2.80GHz	Red Hat Enterprise Linux WS release 3	23
Intel® Pentium® 4 CPU 2.66GHz	Red Hat Enterprise Linux WS release 3	5
Quad Intel® XEON™ MP CPU 1.90GHz	Red Hat Enterprise Linux WS release 3	1
AMD Athlon™ XP 2.08 GHz	Microsoft Windows XP	1

— Large number of participant nodes.

Scheduling in the large scale projects currently handled by the BOINC architecture allows clients to request work based on user preferences. These preferences are on the scale of days of work, and current BOINC projects such as SETI@home are in no danger of running out of workunits to process. The algorithm in the standard BOINC distribution can be seen as a first come, first served algorithm where the client can determine the amount of work served.

3.2. BOINC Scheduling Alternatives

Three new BOINC scheduling alternatives are used in this experiment to measure the performance benefits offered to the calculation by the architecture. The first algorithm is a naïve first come, first serve approach serving a single workunit to nodes (FCFS-1). The second algorithm is a naïve first come, first serve approach serving five workunits to nodes (FCFS-5). The final algorithm is a scheduler based on the ant colony heuristic.

The first come, first serve algorithms allocate either one (FCFS-1) or five (FCFS-5) workunits to a participant node during each work request. Both algorithms account for the current workunits in progress on a node by subtracting the current workunits in progress from the workunit allocation. When a node contacts the scheduler to request work, the algorithm allocates enough workunits to bring the node workload back to the target level according to either FCFS-1 or FCFS-5 target workloads. FCFS-1 results in a single workunit being processed completely by a node before receiving more work. FCFS-5 allows a node to work on five workunits at a time. If a node contacts the scheduler before it completes all five workunits, the scheduler is allowed to send additional workunits to make the total workload five workunits.

The ant colony algorithm is a heuristic based on a colony of ants using pheromone trails to optimize retrieval of food from a food source and returning it to the colony [7]. The scheduling algorithm uses the ant-colony model as a heuristic for determining workunit assignment to participant nodes. The algorithm can be described in terms of the ant-colony metaphor as: the computational resources are the food sources; the computational power is the food; and the job workunits are the ants. The workunit "ants" leave pheromone trails by leaving the server complex and traveling to the computational nodes and returning results back to the server complex. These trails degrade over time so workunits must continually travel to the computation node to reinforce the pheromone level. These pheromone trails influence future workunit "ants" by increasing the probability of choosing the computational resource path. Over time, the computational resources which provide the quickest turn around on workunit computation will attract the most workunits.

The ant colony scheduling technique is a combination of resource discovery, task mapping, and fault tolerance. The use of pheromone levels to determine the performance of participant nodes acts as a dynamic resource discovery mechanism. This metric is then used to assign available tasks during the mapping phase. By comparing pheromone levels of different nodes, the algorithm can map an appropriate proportion of the available work. On the other hand, since the algorithm uses pheromones as a probabilistic heuristic, the algorithm can occasionally test increasing and decreasing task assignment in order to reevaluate a node as more or less powerful based on performance. Finally, since all of the metrics of computational power, communication attributes, node parallelism, architecture, and reliability are consolidated into the pheromone metric, the scheduler can react to fluctuations in any of the metrics. This unified metric decreases over time. Due to this, drops in performance due to faults, communication congestion, or other factors will lead to decreased pheromone levels and proportionally fewer task assignments.

3.3. Participant Nodes

The sequence alignment experiment is executed on a BOINC implementation using a variety of available resources. The environment has 30 participant nodes. The breakdown of processor configurations, host operating systems, and number of each node are shown in Table 1.

The 28 Pentium 4 workstations reside on a local network connected to the same campus network as the

Quad Xeon workstation. The campus network contacts the BOINC central server via the Internet. The AMD Athlon workstation resides on the BOINC server network (Figure 5). All participant nodes use an unmodified BOINC core client version 3.20. The BOINC server software version 3.20 runs on a single host with modifications for the implemented scheduling algorithms.

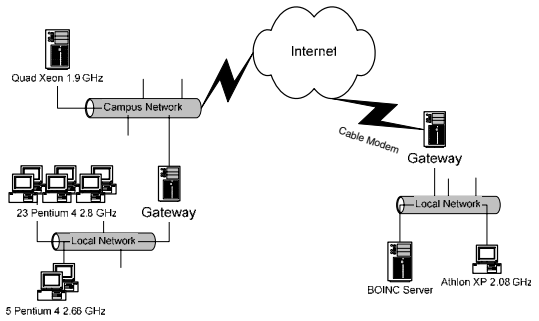

Figure 5 Network Diagram

4. Sequential Algorithm

The sequential algorithm uses a dynamic programming approach to calculate the score for all possible alignments simultaneously [2]. The algorithm produces an alignment matrix as its final output with the scores for each alignment in each cell of the matrix. The alignment matrix takes each base of the first gene sequence as a column heading and each base of the second gene sequence as a row heading:

Sequence 1

Sequence 2	A	A	C	G	T	T	A	G
A								
A								
G								
T								
T								
A								
G								
G								

Figure 6 Inital Alignment Matrix

The algorithm then begins at the upper left corner of the matrix. The base from the column heading is compared to the base of the row heading. A match is given a score (1) while a mismatch is given a penalty (-1). This match adjustment, m, is combined with the score from the upper-left neighbor. Additionally, the scores from the upper neighbor and left neighbor are each combined with a gap penalty (-2). The score, $n_{i,j,}$ for row, i, and column, j, is calculated with the formula:

$$n_{i,j} = \max\left(0, \left(n_{i-1,j-1} + m\right), \left(n_{i-1,j} - 2\right), \left(n_{i,j-1} - 2\right)\right)$$

The resulting scores express the fitness of a particular alignment based on the gap and matching penalties set in the calculation.

Sequence 1

Sequence 2	A	A	C	G	T	T	A	G
A	1	1	0	0	0	0	1	0
A	1	2	0	0	0	0	0	0
G	0	0	1	1	0	0	0	1
T	0	0	0	0	2	1	0	0
T	0	0	0	0	1	3	1	0
A	1	1	0	0	0	1	4	2
G	0	0	0	1	0	0	2	5
G	0	0	0	1	0	0	0	3

Figure 7 Complete Scoring Matrix

Once all the scores are calculated, the matrix can be searched for the highest score and trace back through the neighboring values to find the best sequence alignment. The dotted line in Figure 7 represents the alignment from the previous example. The starting score 5 differs from the example by the -2 from the trailing G base that is unmatched at the end of the sequence. Following the dotted line, a diagonal direction represents a matching sequence alignment. Moving left horizontally represents a gap in sequence 2, while moving upward vertically represents a gap in sequence 1.

Based on the score formula, the score for an entry in the alignment matrix depends on the scores calculated for its previously calculated neighbors. The following diagram illustrates the dependencies of this calculation:

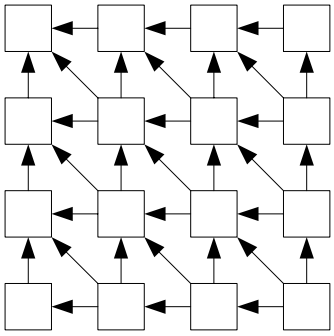

Figure 8 Sequence Alignment Calculation Dependencies

The squares in the diagram represent the cells of the alignment matrix whereas the arrows represent the dependencies of the cell with its neighbors. The top and left border cells assume a zero score for the nonexistent neighbors. The sequential algorithm can progress through the calculation of cells either in row-major, column-major, or diagonal-major order to satisfy the dependencies of each calculation (Figure 9).

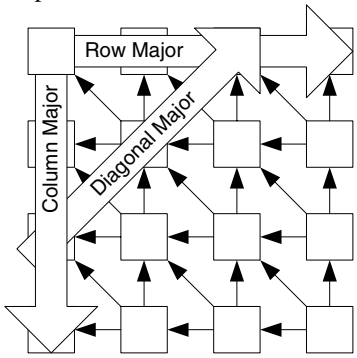

Figure 9 Sequence Alignment Task Order

The sequential performance measures used in this experiment are extrapolated from the calculations needed for a subset of the entire problem. This experiment aligns two generated sequences each of length 100,000. Because the theoretical sequential program would benefit from performing the entire calculation in memory, extrapolated results are acceptable because this theoretical maximum performance would require a machine with 37.3 GB of memory in order to store the entire solution matrix if each cell required a 32 bit integer. Even with memory reduction techniques for storing the sparse solution matrix, the memory required would be many gigabytes. The extrapolated sequential benchmark times only account for the generation of the solution matrix and do not include searching the matrix for the final sequence alignment.

5. BOINC Implementation

The parallel BOINC implementation of the sequence alignment algorithm divides the solution matrix into equally sized sub-matrices and calculates the solutions on the participant nodes. To handle the dependencies of adjacent sub-matrices, the server complex relays the adjacent columns and rows to new compute nodes as they are completed. The final solution matrices are not transferred back to the central server. The experiment implemented for this study does not search the solution matrix for the best alignment. The focus of the experiment is on the task dependency of generating the solution matrix. While returning to the solution matrix to find the final alignment is an important aspect of the calculation, the version of the BOINC software used in the experiment lacked a feature under development which would allow leaving the solution matrix on the client nodes. The missing feature of persistent file storage on client nodes would allow the BOINC implementation to leave the solution matrix on the compute node and send later tasks for alignment retrieval back to nodes containing the appropriate solution segment. While searching these solution matrices are not explored, the solution matrix is written to disk on the client in order to account for disk write times in the benchmarks.

5.1. Workunits

Each workunit represents a sub-matrix of the final solution matrix. The final solution matrix of dimensions 100,000 by 100,000 is divided into 2,500 equal parts using sub-matrices of dimensions 2,000 by 2,000. The input file contains the position of the sub-matrix in the final solution matrix, and the dimensions of the sub-matrix.

Three additional files are included in each workunit containing: the values of the column left-adjacent to the workunit, the values of the row top-adjacent to the workunit, the value of the corner top-left-adjacent of the workunit (Figure 10).

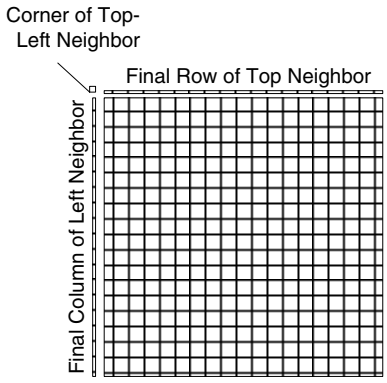

Figure 10 Workunit Input

The gene sequences are not included in each workunit download. Instead, both sequences are transferred in their entirety with the client program. This reduces the redundant data transferred to the client as each workunit will reuse this data throughout the application.

5.2. Result Units

As stated previously, the full alignment matrix is not included with the results sent back to the central server. The results are written to disk on the client node, and required for completion of a workunit. To reduce communication costs, the result unit sends back only the right column, bottom-right corner cell, and bottom row to the server (Figure 11). These three outputs are necessary for the server to generate the new workunits dependent on the current workunit.

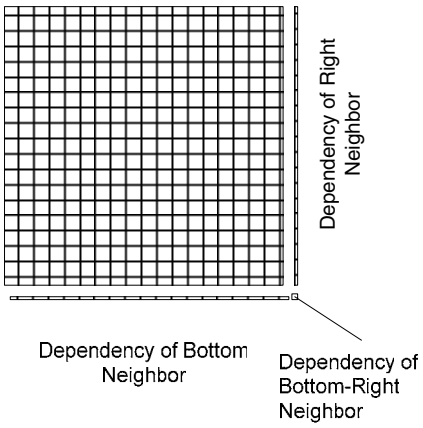

Figure 11 Result Unit Output

5.3. Server Components

While most of the server components are standard, the assimilator plays a central role in the algorithm due to the inter-task dependencies of the algorithm. The web interface, file upload handler, transitioner, and file deleter were unchanged from the standard distribution. The cgi RPC handler and feeder were modified versions of the software used in all three experiments to implement the ant colony scheduling algorithm.

The assimilator of the sequence alignment experiment generates new workunits as the required dependencies are satisfied. Upon receiving the results of a completed workunit, the assimilator checks its records for required inputs of neighboring workunits. If a workunit is found with all three required inputs (top-adjacent row, left-adjacent column, and top-left adjacent corner,) the assimilator generates the new workunit input file and transfers the required inputs to the download directory of the server. Compute nodes contacting the scheduler can then download these input files and the new workunits.

6. Performance Results and Analysis

Task dependency of the calculation produces interesting speedup results due to the amount of parallelism available in the problem initially growing to a peak and then dwindling back to sequential. The structure of task dependency shows differing amounts of speedup for all three scheduling algorithms at different points of the calculation.

Examination of the results must be viewed in the context of the extrapolated sequential running times. These times are based on the best CPU time of a single workunit sized alignment. This time is extrapolated to 2500 workunits for total execution of the problem sequentially for a total runtime of 5000 seconds. This extrapolated time would require in-memory computation of the entire problem in 37.3 GB of memory. A realistic sequential execution would turn to secondary storage increasing the run time. The parallel runtimes represent all computation and communication costs associated with an actual execution.

6.1. Runtime Comparison

Figure 12 shows the parallel runtimes of each scheduling approach versus the extrapolated sequential runtime:

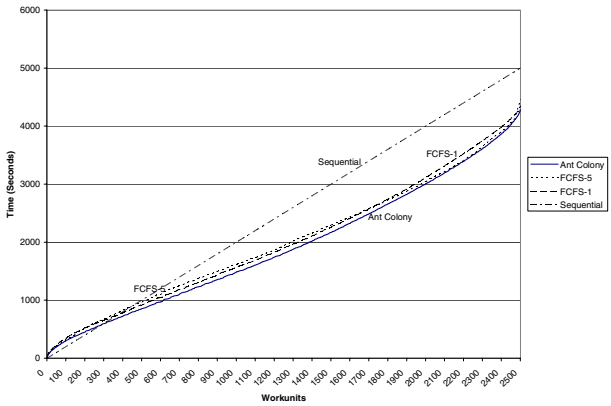

Figure 12 Parallel Runtime Comparison

The runtimes of all three algorithms follow a slight sinusoidal curve due to the task dependency of the sequence alignment problem. Initially, the task dependencies dictate little parallelization available to the system. Tasks must be completed in diagonal major order in order to satisfy the dependencies for later calculations. At step one in the calculation, only one workunit is available. Step two offers two workunits; step three offers three; and so on. This trend continues until the largest diagonal of the matrix is reached and then the number of tasks that can be completed in parallel then starts to increment downward until the lower right corner of the matrix is reached. The parallel implementation can process each of these diagonals in parallel. The first portion of the sinusoid in these runtime curves represents the downward trend of runtimes due to the increasing parallelization. The upward curves of the last portion of the sinusoids represent increasing runtimes due to decreased parallelization.

A close look at the side-by-side runtime comparison shows the FCFS-5 algorithm has the steepest curves at both the beginning and end of the calculation due to the larger blocks of workunits on individual nodes exacerbating the problem of reduced parallelization due to task dependency. However, the differences in the curves are somewhat negligible due to the task generation mechanism of the parallel implementation. New tasks are generated as nodes complete old tasks and report results. The server then accepts the result and generates all tasks with satisfied dependencies. This results in a very small pool of workunits to offer nodes contacting the scheduler. For completed tasks, the scheduler can generate zero, one, or two new tasks. If participating nodes are constantly requesting more work from the scheduler, the likelihood of the scheduler having more than one or two tasks available for scheduling is low. Due to this phenomenon, all three algorithms will perform similarly to FCFS-1 because of the high likelihood of there only being one workunit to schedule. The speedup comparison illustrates more of this phenomenon.

6.2. Speedup Comparison

Figure 13 shows the parallel speedup versus sequential for all three scheduling algorithms:

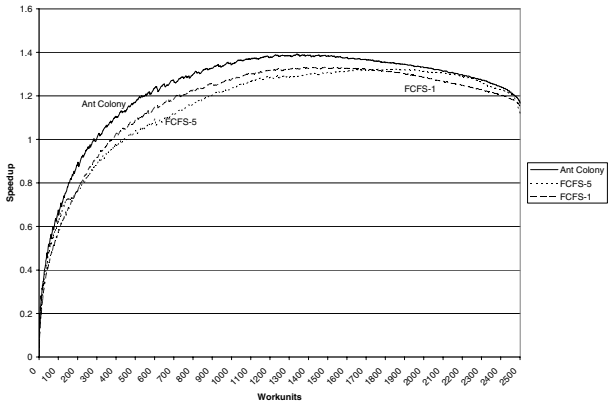

Figure 13 Parallel Speedup

The speedup comparisons again show some interesting although subtle phenomenon. Looking closely at the speedup curves of the three algorithms, all three show slightly different performance peaks and shapes. All three have peaks roughly near the center of the computation due to maximum potential parallelization. The conservative FCFS-1 algorithm shows a steady increase toward the performance peak nearly dead center of the computation. Ant colony features a peak somewhat closer to the beginning of the computation. FCFS-5 shows a slower progression to a peak closer to the end of the computation and a drop-off at the end of the calculation.

The FCFS-1 curve represents the conservative solution to the scheduling problem serving single workunits as they are available according to the task dependency of the calculation.

FCFS-5 represents a more aggressive scheduling approach where most initial diagonals are executed sequentially by single nodes. Take the first few diagonals as an example. When a node completes the first calculation, two new workunits are generated. With FCFS-5, the next node to contact the server would likely receive both workunits. These workunits are run to completion and returned to the scheduler. The next node will, again take all three workunits. This batching of workunits leads to reduced parallelization available for other idle nodes resulting in smaller speedups. On the other hand, this approach also generates larger pools of available workunits, more quickly. Toward the middle of the computation, large

pools of workunits are available for the nodes. Nodes each contact the scheduler and have workunits available leading to greater parallelization later in the calculation.

The ant colony scheduling exhibits different behavior. Because the ant colony scheduler operates blocks limited by the pool of available workunits, it must initially function similarly to FCFS-1. It can only schedule the single workunit available to a requesting node. Additionally, the small amount of parallelization during the beginning of the computation leads to most nodes having very low or non-existent pheromone levels. Later in the beginning portion of the computation, larger pools of workunits become available. Pheromone levels are likely wildly varying due to the randomness of encountering available work during the initial portions of the computation. Occasionally, the stochastic decision making of the ant colony algorithm will choose to send multiple workunits to a node either due to sheer chance or due to a node with an abnormally high pheromone level from the sheer coincidence of a workunits being available at each contact with the server. This process results in the algorithm performing as FCFS-1 for the most part with an occasional sending of multiple workunits. This leads to a buildup of available workunits which leads to greater parallelization early in the computation. Once a large pool of workunits is available, the algorithm will then perform normally allocating blocks of workunits to nodes according to pheromone level. After the major diagonal of the matrix is passed, the available workunits will dwindle, and the now aggressive ant colony algorithm will perform similarly to FCFS-5 where scheduling blocks of workunits leads to eventual starvation of nodes requesting work toward the end of the computation.

7. Conclusions

The public computing implementation of the gene sequence alignment problem illustrates the benefits of the architecture in harnessing not only idle computing cycles of participating nodes but also memory and secondary storage. The ability of public computing to easily aggregate these resources lends itself to calculating problems such as gene sequence alignment which feature $O(mn)$ memory growth. While the speedup offered due to parallelization is modest, the sequential runtimes are extrapolated from the sequential computation time of a portion of the final solution matrix. If this extrapolation were to be realized for the computation of comparing two sequences of 100,000 bases used in this experiment, a machine with 37.3 GB of memory if no memory reduction techniques for storing the solution matrix are used. Future research should include a direct comparison with speeds of a parallel implementation on a small cluster and a sequential implementation.

The performance comparisons of the three scheduling algorithms do expose some interesting phenomenon of the parallel implementation. The ant colony scheduling algorithm expresses some interesting emergent behavior from its scheduling approach. The algorithm initially schedules workunits conservatively in a similar fashion to FCFS-1. This conservative scheduling has the additional factor of occasionally sending multiple workunits to some of the nodes. This results in a quicker buildup of available workunits leading to increased, early parallelization. The ant colony algorithm, however, does not continue its conservative scheduling later in the calculation leading to a quick depletion of available workunits. Overall, it does offer the greatest peak performance of all three algorithms with a 1.4 times sequential speedup midway through the calculation.

8. References

[1] David Sankoff, "The early introduction of dynamic programming into computational biology", *Bioinformatics*, vol. 16 no 1, 41-47, 2000.

[2] T. F. Smith and M. S. Waterman, "Identification of Common Molecular Subsequences," *Journal of Molecular Biology*, 147(1): 195-197, 1981.

[3] D. Anderson, "Public Computing: Reconnecting People to Science," *Conference on Shared Knowledge and the Web*, November 2003.

[4] A. Rosenberg, "On Scheduling Mesh-Structured Computations for Internet-Based Computing," *IEEE Transactions on Computers,* Vol. 53, No. 9, September 2004.

[5] C. Huang, Y. Zheng, and D. Chen, "A Scheduling Approach with Respect to Overlap of Computing and Data Transferring in Grid Computing," *Second International Workshop on Grid and Cooperative Computing*, December 2003.

[6] J. Frey, T. Tannenbaum, and M. Livny, "Condor-G: A Computation Management Agent for Multi-Institutional Grids," *Cluster Computing*, 5(3):237-246, 2002.

[7] M. Dorigo and G. Di Caro, "The Ant Colony Optimization Meta-Heuristic," *New Ideas in Optimization*, McGraw-Hill, 1999, pp. 11-32.