TurboBLAST®: A Parallel Implementation of BLAST Built on the TurboHub

R.D. Bjornson, A.H. Sherman*, S.B. Weston, N. Willard, J. Wing
TurboGenomics, Inc.

Abstract

BLAST (Basic Local Alignment Search Tool) is by far the most widely used application for rapid screening of large sequence databases. This paper describes TurboBLAST, a parallel implementation of BLAST suitable for execution on networked clusters of heterogeneous PCs, workstations, or Macintosh computers.

1. Introduction

For biological scientists, the characterization of novel DNA and protein sequences is an extremely important problem. One powerful approach is based on detailed assessments of the similarity or homology between novel sequences and sequences in large databases of previously characterized genes or proteins. Both functional and evolutionary information may be inferred from when such assessments are used carefully. Unfortunately, detailed similarity assessments may be quite difficult and time consuming to obtain, often requiring compute-intensive structural comparisons or human intervention by a skilled biologist at some point in the process. For this reason, it is common to use simpler techniques to screen large databases to identify a modest number of sequences most likely to be worthy of detailed examination.

BLAST (Basic Local Alignment Search Tool) [1-3,9,10] is by far the most widely used application for rapid screening of large sequence databases. The inputs to BLAST are a set of input query sequences and a number of DNA or protein databases. For each input query sequence, BLAST determines a group of sequences in the databases that have high-scoring pairwise alignments to the query sequence, where the scoring of alignments is based on the use of a user-specified scoring matrix accounting not only for regions of exact matches, but also for insertions, deletions, and substitutions of DNA bases or amino acids. Since the BLAST algorithm considers both local and global alignments, it can detect regions of high similarity embedded in otherwise unrelated sequences, often identifying sequences that, upon detailed examination, can provide important clues about the nature and function of the query sequence.

The National Center for Bioinformatics Information (NCBI) develops and distributes an implementation of BLAST that has become something of a "gold standard" for similarity assessment [2]. When run on modern PCs or workstations, it can process large sequence databases quite quickly. However, sequence databases are exploding in size, growing at an exponential rate that far exceeds the rate of increase in hardware capability (which generally obeys Moore's Law). As a result, the use of the NCBI BLAST application on a single processor has become too costly, inefficient, and time-consuming for many life science laboratories.

To address this problem, we and others have examined the use of parallel computation (e.g., [5,6,13,15]). NCBI itself provides an option that employs multiple threads to accelerate performance on small shared-memory multiprocessors. The common wisdom, and our own experience, is that this works reasonably well for small numbers of processors (2 or 4), but that it does not scale up very well, particularly when multiple databases and small query sequences are used. Others, most notably SGI [6], have developed alternative versions of BLAST that address some of the performance issues in NCBI's threaded implementation, but such alternatives diverge from NCBI's code (leading to significant validation questions) and are difficult to keep in synch with the relatively frequent updates provided by NCBI.

Even a "perfect" threaded implementation of BLAST, however, would have to face the fact that database growth will require the use of increasingly large and costly multiprocessor machines. Given the widespread availability of powerful, yet cheap commodity PCs, a potentially superior alternative is the use of networked clusters of such machines. To pursue this alternative, we have developed TurboBLAST [15], an accelerated, parallel deployment of NCBI BLAST. We use the term "deployment" rather than "implementation" because TurboBLAST delivers high-performance not by changing the BLAST algorithm, but by coordinating the use of multiple copies of the unmodified serial NCBI BLAST application. As a result, TurboBLAST supports all of the standard variants of the BLAST algorithm supported in NCBI BLAST (blastn, blastp, blastx, tblastn, and tblastx), it provides results that are effectively identical to those obtained with the NCBI application, and it is easy to keep up with new versions of BLAST as they are distributed by NCBI. TurboBLAST is available for

*Contact author.

many parallel computing environments, from heterogeneous clusters of PCs, workstations, and Macintosh computers, to parallel supercomputers, to the worldwide computing grid. All that is required is that each machine have a Java virtual machine (JVM) and a native executable for NCBI's BLAST application.

In the remainder of this paper, we first provide background on sequence similarity searching in general and provide relevant details about BLAST. Then we turn to TurboBLAST, first describing our general parallelization strategy and then going on to discuss the implementation of TurboBLAST, the underlying TurboHub system on which it is built, and some of the details of the approach to task and database splitting that is among the most important aspects of TurboBLAST. Finally, we conclude by providing a sampling of benchmark results to illustrate the performance achievable with TurboBLAST.

2. Sequence Similarity Searching and BLAST

2.1 General Background

In this paper we are concerned with comparisons among sequences of letters representing members of two classes of biological sequences: nucleotide sequences, in which each letter represents one of four DNA bases, and peptide or protein sequences, in which each letter represents one of twenty amino acid residues. There are many different methods for comparing such sequences. Some methods, such as those based on the analysis of transformational grammars (cf. [9], Chapter 9), compare sequences by comparing the properties of the mathematical algorithms that may be used to generate the sequences in question. However, most commonly used methods involve the use of direct sequence alignment at some point in the comparison process.

Sequence alignment provides an explicit mapping between the letters in two or more sequences. For the purposes of this paper, we will consider only pairwise alignment involving just two sequences. Given a pair of sequences, there are many possible alignments, and each one can be assigned a quality score using a metric that rewards exact letter matches and that penalizes substitutions (i.e., where a letter in one sequence is mapped to a different letter in the other sequence) and gaps (i.e., where mapped letters are the same, but they occur in different relative positions due to insertions or deletions in the sequences). The rewards or penalties for matches and substitutions are typically specified in the form of a 4x4 or 20x20 table (called a "substitution matrix" or "scoring matrix") whose entries reflect the biological significance of the corresponding matches or substitutions. Gaps are penalized using a formula that may depend on the number, position, and length of each gap. The global similarity score of a pair of sequences is simply the score of the best (i.e., highest-scoring) alignment of the entirety of the two sequences. Even if the global similarity score is low, there may still be portions of the sequences that match extremely well, and such local alignments are often of higher biological interest than the best global alignment.

To assess the similarity of a given input query sequence to an entire database, the query sequence is aligned separately to each sequence in the database in order to identify those database sequences with the highest-scoring local alignments (called "hits"). The results of this process (often called a "similarity search") are then reported as a rank-ordered hit list, often augmented by a series of individual sequence alignments and various overall scores and statistics.

There are numerous programs and algorithms available to perform similarity searching. For a basic discussion of bioinformatics and sequence similarity searching, see [4] and [7]. One of the earliest algorithms for performing sequence similarity searching using pairwise alignment was implemented in the FASTA program [11,12].

2.2 The BLAST Algorithm

BLAST [1-3,9,10] is certainly the most popular algorithm for sequence similarity searching. The approach used by the BLAST algorithm is to first identify short segments with high-scoring alignments without gaps, and then to extend each such local alignment as far as possible in both directions, with or without gaps, so long as the score resulting after each new extension remains sufficiently large. The method then evaluates the statistical significance of all such high-scoring matches and reports as hits only those that satisfy a pre-selected threshold of significance.

More precisely, BLAST begins by dividing the input query sequence into all possible contiguous subsequences (called "words") of length w (called the "word length"). The value of w depends on the type of sequence involved, but defaults to w=3 for comparison of protein sequences, for example. For a given database sequence, BLAST searches for subsequences that exactly match one of the words. When such a match is found, the search process is suspended, and BLAST tries to extend the subsequence match in both directions (possibly introducing gaps), so long as the score for the extended match does not decrease significantly from the score for the original word match. Ultimately, the extension process terminates either when the end of one of the sequences is reached, or when the score has diminished sufficiently. If the score at that point is high enough, then the extended match is tentatively included in the hit list (possibly displacing a hit that was found earlier), and the search for word

matches resumes. When all word matches have been processed for a given database sequence, that sequence is discarded, and work starts on the next sequence in the database. At the end of the entire process, BLAST reports the hit list along with various overall statistics. The amount by which the local score may diminish before terminating the extension process, and the threshold for inclusion in the hit list are among a number of empirically-determined parameters that are supplied as inputs to BLAST.

The implementation of BLAST from NCBI can perform five different types of similarity searches, corresponding to different combinations of sequence types in the input queries and databases. The simplest ones compare a nucleotide or peptide sequence with sequences of the same type. The blastp program compares a peptide query sequence against a database of protein sequences. The blastn program compares a nucleotide query sequence against a database of nucleotide sequences.

The other three types of searches compare sequences are more complex. In living cells, there is a relationship between nucleotide and peptide sequences in which each sequence of three consecutive nucleotide bases (called a "codon") is translated into a corresponding amino acid. In general, for a given nucleotide sequence, it is impossible to know where to start marking off the codons. Thus comparisons must consider three possible translations corresponding to different "frames" that start marking off the codons at the first, second or third position in the sequence. Since DNA has two strands, and each of the three frames determined by the nucleotide sequence in question has an analogue on the complementary strand that is translated in the opposite direction, there are a total of six frames that must be considered. The blastx program compares the six-frame conceptual translation products of a nucleotide query sequence (both strands) against a protein sequence database. The tblastn program compares a protein query sequence against a nucleotide sequence database in which each sequences is dynamically translated in all six reading frames (both strands). Finally, the tblastx program compares the six-frame translations of a nucleotide query sequence against the six-frame translations of a nucleotide sequence database.

The program blastall is provided by NCBI as single interface permitting access to all five types of BLAST comparisons. It permits a user to specify a group of input query sequences and one or more databases against which each of the query sequences is to be compared. For the purposes of running the algorithm and computing overall statistics, blastall treats multiple databases as if they were aggregated into a single large "virtual database" that is mapped to the available virtual memory space on the machine performing the search. Each query sequence is searched against the entire virtual database before moving on to the next query sequence.

The computational cost of applying the BLAST algorithm to a pair of sequences is clearly linear in the lengths of the sequences, although the constant of proportionality may vary widely for different databases and query sequences. (As a result, it should not be assumed that searching against half of a database takes even roughly half the time required to search against the entire database.) While the description above makes the computation seem quite tedious and lengthy, in practice, the pairwise comparison process has been made very efficient through the use of a binary encoding of the sequences that reduces memory requirements substantially and permits use of a clever implementation based on finite automata. To extend the pairwise comparison to a complete search of one query sequence s against a number of databases $D_1, D_2, D_3, \ldots, D_n$ blastall uses an iteration similar to the following:

```
For each Query Sequence s {
  For each Database Di {
    For each sequence d in Di {
      Compare s to d using BLAST;
      Update aggregate statistics}}
    Report results for sequence s}
```

As noted above, the entire contents of the individual databases are aggregated and mapped as a single virtual database onto the available virtual memory. When the mapped database can be held entirely in available physical memory, the iteration runs quite efficiently. However, a difficulty arises when the size of the virtual database exceeds the available physical memory. In that event, a significant amount of paging may occur as portions of the virtual database are brought into physical memory from disk, replacing other previously loaded data. Not surprisingly, this introduces a significant amount of noncomputational overhead and inefficiency. that may, in many cases, far exceed the computational cost that would have been incurred by the original BLAST computation had there been sufficient physical memory available to hold the entire database.

3. The TurboBLAST System

3.1 Strategy

An individual BLAST job specifies a number of input query sequences to be searched against one or more sequence databases. In order to achieve parallel speed-up, TurboBLAST implements a distributed Java "harness" that splits BLAST jobs into multiple small pieces, processes the pieces in parallel, and integrates the results

into a unified output. The harness coordinates the following activities on multiple machines:

- Creation of BLAST tasks, each of which requires the comparison of a small group of query sequences (typically 10-20 sequences) against a modest-sized partition of one of the databases sized so that the entire task can be completed within the available physical memory without paging;
- Application of the standard NCBI blastall program to complete each task; and
- Integration of the task results into a unified output.

This approach has the advantage that it is guaranteed to generate the same pairwise sequence comparisons as the serial version of BLAST since it uses exactly the same executable to perform the search computations. High performance is achieved in two ways. First, the size of each individual BLAST task is set adaptively so that blastall processing will be efficient on the processor that computes the task. Second, a large enough set of tasks is created so that all the processors have useful work to do and so that nearly perfect load balance can be achieved.

In the implementation described in the next section, the task creation process occurs in two parts. First, at the time of job submission, we create "initial tasks" that search a group of 10-20 sequences against all of the database(s). Later, if any the initial tasks is too large for efficient blastall processing on the machine that is working on it, the task is split dynamically into smaller subtasks by partitioning the database(s) in order to create tasks that are small enough for that machine. We discuss the details of task splitting below after first describing the overall implementation.

3.2 Implementation

To implement our approach, we have designed a classic three-tier system comprising three principal components: a Client, a Master, and a number of Workers. TurboBLAST is delivered with standard versions of the Client and Master components, but either may be customized to meet the needs of individual situations.

Client: The Client is the part of the system accessed by an end user submitting BLAST jobs. The Client takes a BLAST job and divides it into a number of initial BLAST tasks, each of which searches a small group of the input query sequences against the full set of databases for the job. The Client submits these initial tasks to the Master, retrieves the results when they are available, and writes the results to a single file in the proper order. Currently, two Clients are available: one that provides a web-browser interface similar to the one available at the NCBI web site, and another designed for command-line use via a tblastall command that is the TurboBLAST equivalent of blastall.

Master: The Master is a Java application that accepts initial BLAST tasks from one or more Clients and sets them up to for processing by the Workers. The Master includes the server portion of a sophisticated parallel execution system called the TurboHub that manages task execution, coordinates the Workers, and provides a virtual shared memories (VSM) used to share data among all the various components. The TurboHub supports dynamic changes in the set of Workers, fault tolerance, and other features that are essential in a robust software application intended for use in commercial settings.

Another important part of the Master is the File Provider, a Java application that manages the genomic databases used for TurboBLAST jobs. The File Provider maintains a system-wide canonical copy of each database, and it delivers all or part of each database to the BLAST Workers as they require them. (Essentially, the BLAST Workers may thought of as having local caches on disk of portions of the canonical databases, and they update their cached copies automatically as required.) While the VSM is used to share and exchange a variety of data involved in a TurboBLAST run, the large files representing databases are transmitted from the Master to the Workers using a high-performance direct transmission protocol that does not involve the TurboHub server.

Workers: Workers are processors that run a Java application that performs the actual BLAST computations by instantiating a local copy of the standard NCBI blastall application (a compiled C executable). As noted in the last section, blastall is only efficient when applied to BLAST searches for which the memory-mapped database(s) fit into available physical memory. If all the available tasks are too large, then the Workers will create smaller subtasks by partitioning the database(s) until the resulting subtasks are small enough. As the computation proceeds, some of the Workers will merge the subtask results to create the final results for each of the initial tasks. This entails parsing the blastall output (stored as XML data) and may require rescaling the scores and statistics to reflect the aggregate size of the databases in the initial task instead of the databases in the subtasks.

An important aspect of Worker operation is scheduling, a topic that we have insufficient space to address here in any detail. The Piranha model [8] used by TurboBLAST is based on the use of a decentralized distributed scheduling system in which each participating machine makes its own scheduling decisions. In the case of TurboBLAST, the Workers make use of a scheduling algorithm to decide what tasks to perform, when to switch to merging, how to balance the potentially competing goals of efficiency and fairness when there are multiple TurboBLAST jobs competing for attention.

3.3 The TurboHub System

At the heart of TurboBLAST is the TurboHub, an execution engine for parallel and distributed Java applications developed by TurboGenomics and based, in part, on the Paradise® system for distributed computing developed by Scientific Computing Associates [14]. The TurboHub is capable of delivering scalable high performance in a wide range of computing environments, from heterogeneous networks of PCs, Macs, or UNIX/Linux workstations, through multiprocessor parallel servers, to the "Computational Grid" formed from wide-area networks of diverse machines.

While the TurboHub supports many types of parallel applications, it has been designed and tuned especially to provide automation and dynamic acceleration for data-parallel applications in which large numbers of independent tasks corresponding to computations on independent data must be processed efficiently in parallel. TurboBLAST is one example of such an application, but there is a broad class of applications known as workflows that are ubiquitous in bioinformatics computation. Workflows may be thought of as flowcharts (including logic and loops), where the flowchart boxes (components) correspond to computational applications or database accesses. Applications written in any language (e.g., NCBI's blastall, which is written in C) may be enabled for use as workflow components in the TurboHub simply by embedding them in thin wrappers written in Java following to a published API. The TurboHub manages the flow of data through the workflows, automatically scheduling the components, transforming data as required between them, balancing load, and handling any errors that may occur.

In the case of TurboBLAST, the workflow is extremely simple and contains only a single component (the Worker, which is a wrapped-up blastall component), so the TurboHub has a relatively simple job to do. (It simply starts the Worker component on every available machine.) In general, however, the TurboHub makes use of a Piranha scheduling model to provide support for accelerating workflows in a number of ways:

- **Pipelining**: By default, individual components are scheduled to run on separate processors or machines, with data passed automatically from one machine to the next. This allows all components to operate concurrently, with each one working on a different portion of the data.

- **Component Replication**: The TurboHub recognizes when a slow component becomes a bottleneck in a workflow, and it automatically schedules extra instances of such components to eliminate the bottlenecks. The TurboHub dynamically manages the flow of data to the multiple component instances to ensure that the load is balanced evenly among them.

- **Parallel Components**: A wide range of applications may realize performance benefits from the use of parallel computing in which multiple processors are all focused on the solution of a single instance of an application. Many of these applications (such as TurboBLAST) may be sped up by preprocessing their input data, carrying out a large number of independent, concurrent computations on the preprocessed data (in parallel, and in any order), and post-processing the results of the independent computations to construct the final outputs. In such cases, the implementation in TurboHub simply requires creation of a suitable Java wrapper to handle the data manipulations; the TurboHub itself handles the management of the task processing. In more complex cases, parallel applications may be implemented using the TurboHub in combination with tools from SCA or with any of a number of other standard technologies such as MPI, PVM, or OpenMP.

Most of the TurboHub's acceleration of workflows takes place automatically, without any user intervention or modification of the original applications underlying the components. The TurboHub makes dynamic decisions about component scheduling, fault recovery, and other aspects of workflow execution, as appropriate, based, in part, on information available to it from the Java wrappers created to convert the original applications into TurboHub-enabled components in the first place.

3.4 Task/Database Splitting

A significant aspect of TurboBLAST is the technique we use to create suitably-sized BLAST tasks for the Workers. We view task creation as somewhat of a balancing act involving two competing goals. On the one hand, it seems desirable to maximize resource utilization and minimize task startup overhead by having each

Worker process the largest possible tasks for which it can run tblastall efficiently given its available physical memory and other computational resource limitations. In an extreme case, this could mean that a Worker with a lot of memory might take on the entire BLAST job, completely eliminating the potential performance gains to be had from parallel computing. In more ordinary situations, there might be a very small number of tasks per Worker, so that it load imbalance among the Workers might limit the performance gains, particularly if the number of Workers varies over time or if the Workers are heterogeneous with respect to their capabilities.

Taking the opposing view, one might choose to have a large number of relatively small tasks. This would permit more Workers to participate, increasing the potential performance gains. It would also tend to deliver a higher degree of load balance among the Workers, since there would be a finer granularity division of the total amount of work. However, there many down sides to using a large number of small tasks. For example: computational resources may be wasted for small tasks; there is an increase in aggregate task startup overhead (i.e., the cost of communicating the task data and results, and the cost of actually starting the tblastall executable for the task); the effort required to merge the task results will grow; and it is likely that the aggregate amount of network communication cost will go up.

We have adopted an intermediate approach to task creation that seems to work well in practice. As noted above, we start by creating initial tasks that we believe are large enough so that the task startup overhead is negligible, even if the databases are eventually divided into relatively small pieces. Our experience has been that the time for communication and tblastall startup is rarely more than a tiny fraction of the tblastall run time, so long as the tasks have at least 10-20 input query sequences, and the Workers have a reasonable amount of physical memory (say at least 256 Megabytes per CPU).

While creating the initial tasks often provides a sufficient degree of parallelism, it frequently occurs that the initial tasks are too large for the Workers, since each task searches against all of the database(s). When a task is too large, Workers leave the set of query sequences as-is and split the database(s) to create two smaller subtasks. When a task involves multiple databases, the subtasks are created so that roughly half the databases are in each task. When a task involves only a single database, the subtasks are created by partitioning the single database. Workers require only a small amount of data (passed via the VSM) to assess whether a particular task is of suitable size and, if necessary, to determine how to split it into two subtasks. The actual database files are never sent to a Worker until it actually requires them to run a BLAST task using tblastall.

We have experimented with a variety of techniques for actually splitting the databases. At one extreme, we implemented "virtual splitting" in which the database is never split physically, but only a portion of the database is mapped to virtual memory. This has the advantage of generality (since the database may be split dynamically on any boundary between sequences), but it means that every database must be delivered to every machine, consuming significant time and network bandwidth. As an alternative, we also considered storing multiple copies of each database on the Master, corresponding to partitionings of differing granularity (e.g., halves, quarters, eighths, etc.). This avoids the need to deliver all the data to every machine, but it requires that the database server store multiple copies of the databases consume a lot of disk space and are difficult to manage.

In our current implementation, we require that databases be pre-split into a number of partitions using the standard NCBI database formatting program formatdb. Depending on the characteristics of the database, formatdb will create 3, 5, or 7 files to represent each partition. Of these, only one (the file containing the actual sequence data for the partition) is large; the others are relatively small index files. We view the partitions created by formatdb as the leaves in a binary partition tree, and we build "alias files" that enable us to represent the databases corresponding to any of the nodes in the tree. Specifically, the database corresponding to any given node is composed of the single alias file for the node plus all of the files created by formatdb for the leaves in the sub-tree rooted at the node. When a Worker needs to obtain a particular portion of a database, it makes a request to the File Provider, which responds by delivering the necessary files to the Worker.

Our current approach has a number of advantages. First, it is reasonably general, since individual databases may be divided at a number of different granularities. Moreover, since there is no requirement that all subtasks use the same partitioning of any database, Workers can easily adapt the task sizes to their own capabilities. Another advantage is that while the File Provider is able to deliver exactly the required amount of sequence data to each of the Workers, it need not waste disk space by storing more than one copy of each formatted database. (The alias files are quite small.)

4. Performance Results

We conclude this paper by presenting a few benchmark results to illustrate the performance achievable with TurboBLAST. In the first example, 50 Expressed Sequence Tags (ESTs) totaling 18,500 nucleotides were searched against three databases obtained from the NCBI web site. The databases were *Drosophila* (1,170 sequences containing approximately 123 million

nucleotides), the *GSS* Division of GENBANK (approximately 1.27 million sequences containing 651 million nucleotides), and *E-coli* (400 sequences containing approximately 4.6 million nucleotides).

The search was performed using the blastn variant of blastall, and it was run on a group of IBM Netfinity PCs, each containing a single 500-Megahertz Pentium III processor, 512 Kilobytes of cache memory, and 256 Megabytes of main memory. The PCs were connected via a switched 100-Megabit Ethernet network. The serial run required 2131.8 second (wall-clock time) on one of the machines. With 11 Workers, only 130.0 seconds was required, representing a speedup of more than a factor of 16. Times and speedup factors for runs with varying numbers of Workers are shown in Figures 1a and 1b. Clearly the superlinear speedup here is due to the elimination of the paging overhead discussed earlier, rather than to a superlinear reduction in actual computation time. This is an excellent example of a case where the overhead dominates the tblastall computation time, since the TurboBLAST time achieved with just one worker was just over 1000 seconds, nearly twice as fast as serial blastall.

As a second example, we used the blastx variant of blastall to perform a search in which the input queries were chromosomes 1, 2, and 4 from the *Arabidopsis* genome, and the database was the *Swiss-Prot* protein database. In this case the database contained roughly 12.8 million peptides. The search was run on the same set of machines as in the first example, and TurboBLAST was able to achieve a speedup of nearly 10.8 using 11 Workers, reducing the serial time of 5 days, 19 hours, and 13 minutes to a parallel time of 12 hours, 54 minutes.

Finally, our third example involves the use of the blastn variant to perform a search in which the input queries were 500 mouse ESTs of containing 200-400 nucleotides each, and the database was a version of the NT database from NCBI that contained at total of 1,681,522,266 nucleotides. In this case, the benchmarks were run on an IBM Linux cluster containing 8 dual-processor workstations connected via 100-Megabit Ethernet. Each workstation contained two 996-Megahertz Pentium III processors and 2 Gigabytes of physical memory. The serial blastall run required 4945 seconds, and the parallel results are tabulated in Table 1. With Workers running on each of the 8 workstations (using 16 CPUs), the speedup was nearly a factor of 14, representing nearly 90% parallel efficiency, despite the fact that one of two of the workstations were also serving as the Client and the Master, respectively.

5. References

[1] Altschul, S.F. & Gish, W., "Local alignment statistics." **Meth. Enzymol. 266**:460-480 (1996).

[2] Altschul, S.F., *et al.*, "Basic local alignment search tool." **J. Mol. Biol. 215**:403-410 (1990).

[3] Altschul, S.F., *et al.*, "Gapped BLAST and PSI-BLAST: a new generation of protein database search programs." **Nucleic Acids Res. 25**:3389-3402 (1997).

[4] Baxevanis, A.D. and Ouellette, B.F., eds., *Bioinformatics: A Practical Guide to the Analysis of Genes and Proteins*, Wiley-Interscience (1998).

[5] Braun, R.C., *et al.*, in *Future Generation Computer Systems* **17**:745-754 (2001).

[6] Camp, N. "High Throughput BLAST." Technical Report, Silicon Graphics, Inc. (1998).

[7] Durbin, R. *et al.*, *Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids*, Cambridge University Press (1998).

[8] Kaminsky, D., *Adaptive Parallelism with Piranha*, Ph.D. Dissertation, Yale University, 1994.

[9] Karlin, S. & Altschul, S.F., "Methods for assessing the statistical significance of molecular sequence features by using general scoring schemes." **Proc. Natl. Acad. Sci. USA 87**:2264-2268 (1990).

[10] Karlin, S. & Altschul, S.F., "Applications and statistics for multiple high-scoring segments in molecular sequences." **Proc. Natl. Acad. Sci. USA 90**:5873-5877 (1993).

[11] Lipman, D.J. and Pearson, W.R., "Rapid and sensitive protein similarity searches," **Science 227**:1435-1441 (1985).

[12] Pearson, W.R. and Lipman, D.J., "Improved tools for biological sequence comparison," **Proc. Natl. Acad. Sci., Vol. 85**:2444-2448 (1988).

[13] Pedretti, K.T., *et al.*, in *Lecture Notes in Computer Science*, **1662**:271-282 (1999).

[14] Scientific Computing Associates, Inc., *Paradise User's Guide & Reference Manual* (Version 4.0). New Haven, CT, 1996.

[15] TurboGenomics, Inc., *TurboBLAST User's Guide* (Version 1.1). New Haven, CT, 2001.

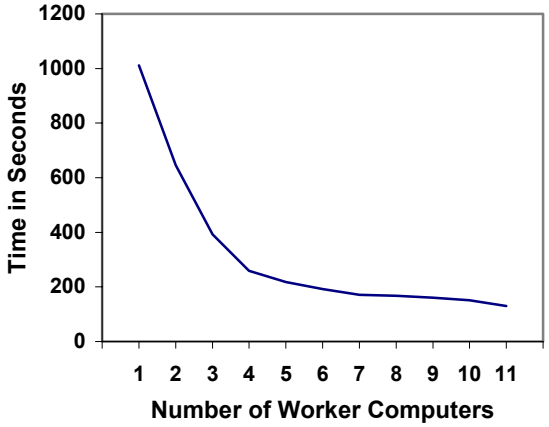

Figure 1a: Times for Example 1

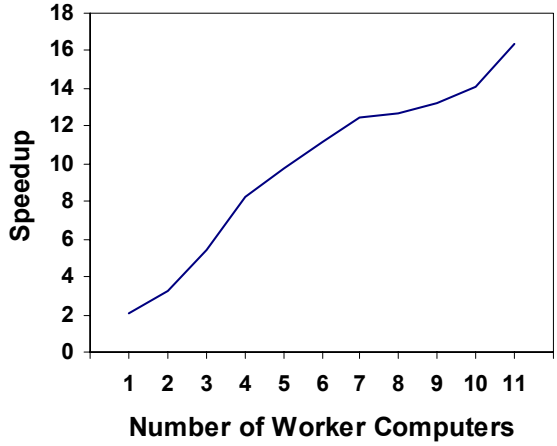

Figure 1b: Parallel Speedups for Example 1

Number of Worker Processors	Time (Secs.)	Parallel Speedup
2	2534.38	1.95
4	1277.19	3.87
6	862.90	5.73
8	658.78	7.51
10	520.67	9.50
12	460.69	10.74
14	403.64	12.25
16	357.03	13.85

Table 1: Results for Example 3