ScalaBLAST: A Scalable Implementation of BLAST for High-Performance Data-Intensive Bioinformatics Analysis

Christopher Oehmen, *Member*, *IEEE*, and Jarek Nieplocha, *Member*, *IEEE Computer Society*

Abstract—Genes in an organism's DNA (genome) have embedded in them information about proteins, which are the molecules that do most of a cell's work. A typical bacterial genome contains on the order of 5,000 genes. Mammalian genomes can contain tens of thousands of genes. For each genome sequenced, the challenge is to identify protein components (proteome) being actively used for a given set of conditions. Fundamentally, sequence alignment is a sequence matching problem focused on unlocking protein information embedded in the genetic code, making it possible to assemble a "tree of life" by comparing new sequences against all sequences from known organisms. But, the memory footprint of sequence data is growing more rapidly than per-node core memory. Despite years of research and development, high-performance sequence alignment applications either do not scale well, cannot accommodate very large databases in core, or require special hardware. We have developed a high-performance sequence alignment application, ScalaBLAST, which accommodates very large databases and which scales linearly to as many as thousands of processors on both distributed memory and shared memory architectures, representing a substantial improvement over the current state-of-the-art in high-performance sequence alignment with scaling and portability. ScalaBLAST relies on a collection of techniques—distributing the target database over available memory, multilevel parallelism to exploit concurrency, parallel I/O, and latency hiding through data prefetching—to achieve high-performance and scalability. This demonstrated approach of database sharing combined with effective task scheduling should have broad ranging applications to other informatics-driven sciences.

Index Terms—High-performance sequence alignment, BLAST, Global Arrays.

1 Introduction

The driving force behind studying newly sequenced genomes is the desire to understand an organism at the level of molecular interactions. Many biological processes involve interaction of proteins, or molecular building-blocks, which can assume an astonishing array of functions; hence, assessing the identity and function of proteins in a newly sequenced organism is critical to understanding the organism's molecular interactions. The function of a protein can often be surmised by its similarity to proteins of known function using a process called sequence alignment. Sequence alignment involves comparing the sequence of known organisms to the nucleotide sequence of DNA or RNA or to the amino acid residue sequence of proteins in the newly sequenced organism. Since DNA, RNA, and proteins can all be represented as linear character strings (hence, the term "sequence"), scoring the similarity of two sequences can be considered in the algorithmic sense as a string matching problem.

Many sequence alignment algorithms have been developed for quantifying the homology of a pair of amino acid or nucleotide sequences. Smith-Waterman [1], Smith et al. [2], Needleman-Wunsch [3], and BLAST [4], [5] are examples of sequence alignment algorithms that have been widely used for research in biology. Of these methods, BLAST is the most computationally efficient and is generally the most widely used, even though it has lower sensitivity than the other methods [6], [7]. The most common access to BLAST is through Web applications or installations of standalone packages. These applications are extremely effective in delivering quick sequence homology scores for small numbers of queries against most databases. However, when a task requires millions of queries to be performed at once, a serial BLAST application may require weeks or longer to complete. The increased need for performing large numbers of sequence alignments in a single task, for instance in multiple genome or multiple proteome comparisons [8], [9], has motivated the development of parallel applications that take advantage of multiprocessor architectures.

ScalaBLAST was developed using a *software-based* shared memory approach with the goal of achieving efficiency and scalability on shared as well as distributed memory architectures. ScalaBLAST design is based on two main concepts: 1) sharing of very large databases among processors to accommodate the exponential growth in the size of publicly available sequence data and 2) retaining the efficient scaling characteristic of conventional query-scheduling applications, a feature which has been problematic in previous database sharing applications, as discussed in more detail in the related work section.

The performance characteristics of ScalaBLAST are illustrated in this paper with experimental results on the

• *The authors are with the Computational Sciences and Mathematics Division, Pacific Northwest National Laboratory, 902 Battelle Boulevard, PO Box 999, MSIN: K7-90, Richland, WA 99352.*
E-mail: {Christopher.Oehmen, Jarek.Nieplocha}@pnl.gov.

Manuscript received 3 July 2005; revised 30 Jan. 2006; accepted 8 Mar. 2006; published online 26 June 2006.
Recommended for acceptance by N. Amato, S. Aluru, and D. Bader.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDSSI-0321-0705.

two classes of computer architectures, the SGI Altix and the HP cluster with Quadrics interconnect. For comparison, we also show results from a serial version of standalone BLAST and two representative high-performance BLAST implementations: MPI-BLAST [10], which is a readily available and widely used open-source implementation of high-performance BLAST, and HTC-BLAST [11], which is highly optimized by SGI for their shared memory architecture of SGI ALTIX.

ScalaBLAST takes advantage of the computational independence of individual queries while mitigating per-processor memory limitations by distributing the database files using the Global Arrays toolkit [12], a shared memory programming interface that can be used on shared or distributed memory architectures. For our benchmark data set, ScalaBLAST scales significantly better than MPI-BLAST and performs as well as HTC-BLAST in terms of runtime for real-world queries against the nonredundant protein database (*nr*). ScalaBLAST's advantage over HTC-BLAST is portability. ScalaBLAST is highly efficient and scalable on both shared memory and distributed memory architectures as it does not rely on hardware implementation of shared memory. The excellent performance and scalability of ScalaBLAST are due to a combination of several optimization techniques that we have found to be broadly applicable to high-performance sequence analysis in general. These techniques include:

1. *Distributing the target database over available memory*: Global array objects are used to store the database sequences in "shared" memory segments available to all processors. Nonblocking operations on global arrays are utilized to exploit nonuniform memory access characteristics by overlapping retrieval of sequence data with computation of the sequence alignment. The goal is to eliminate the need for frequent file access during a query and to enable searching against extremely large databases, which are expected for informatics-driven science.
2. *Multilevel parallelism to exploit concurrency*: This involves scheduling queries to individual process groups to constrain the need for result-merging to fewer processors and to improve the overall concurrency.
3. *Parallel I/O*: I/O performance is improved by allowing each processor to create its own output file.
4. *Latency hiding*: Sequence block prefetching using nonblocking memory access calls to prefill large local blocks of memory with sequence data so that latency associated with getting the data is hidden on both shared and distributed memory architectures.

The remainder of the paper is organized as follows: Section 2 summarizes existing related work. Section 3 describes our motivation and design goals for the parallel implementation of BLAST. Section 4 provides details of the technical approach. In Section 5, we present benchmark results for ScalaBLAST on both shared and distributed memory architectures in this paper with respect to performance of HTC-BLAST on ALTIX, and best runtimes observed for MPI-BLAST on a HP Linux cluster with a high-performance Quadrics Elan-4 interconnect. Section 6 describes early but successful experience with using ScalaBLAST by bioinformatics researchers for grand challenge class of problems. Section 7 includes discussion of the broader context and applicability of the proposed parallelization strategy. Finally, conclusions are given in Section 8.

2 Other Related Work

We present here an overview of key technical aspects of representative applications involving query scheduling or database parsing. Several approaches to high-performance sequence alignment have been developed to accommodate the need for running large numbers of queries in a single batch job. Special hardware solutions for sequence alignment include special chip design to accommodate a particular alignment algorithm [13], or the use of field-programmable gate arrays to increase the speed of alignment calculation [14].

A second approach to increasing throughput of sequence alignment tools is the use of centralized service centers. For instance, Soap-HT-BLAST achieves parallelism by scheduling incoming queries through a Web service that distributes the queries appropriately to available resources [15]. Similarly, TurboHub is a resource for executing parallel and distributed Java applications that can be used in conjunction with TurboBLAST to schedule queries over available processors in a network environment [16]. Integration of sequence alignment with other analysis tools and data sources using a centralized server has increased the rate at which practical knowledge can be extracted from experimental and sequence data [17].

Combining high-performance architectures with specialized software solutions has been shown to improve sequence alignment throughput. SGI has developed an optimized BLAST query scheduler that takes advantage of shared memory hardware to provide significant speedup for sequence alignment [11], [18]. This vendor proprietary implementation is distributed in a binary format and has been optimized for the Altix shared memory architecture. By taking advantage of low-latency memory access and multiprocessors, the throughput of BLAST queries has been dramatically increased over the sequential version of BLAST. Heterogeneous distributed/shared memory architectures have also been exploited by scheduling a prefiltering step over a large distributed system, followed by vectorizing the sequence alignment scoring step [19]. Another hybrid application, ParAlign, uses single instruction, multiple data (SIMD) supported architectures with optimized software to achieve speedup of several of the most popular sequence alignment algorithms [20].

Finally, many software solutions for achieving parallelism for sequence alignment have been put forth. MPI-BLAST is one of the most widely used software packages for achieving high throughput sequence alignment [10]. The main benefit from using MPI-BLAST arises when the fragments are numerous enough (and, therefore, small enough) to fit entirely in memory, thus allowing the application to perform alignments without having to retrieve sequences from a file during the run [21]. Recently, serious performance issues with MPI-BLAST have been identified on shared memory architecture, including a serial bottleneck that occurs during the

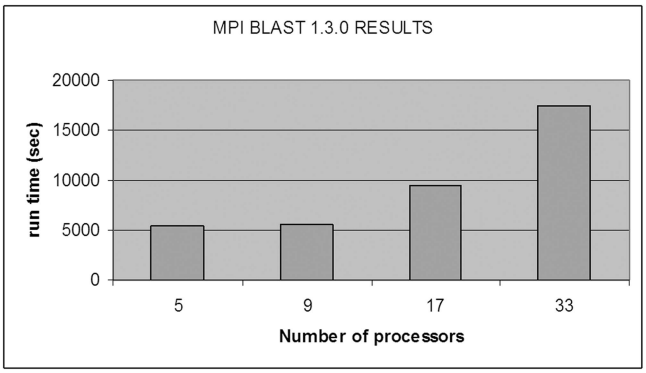

Fig. 1. MPI-BLAST runtimes for 1,000 queries.

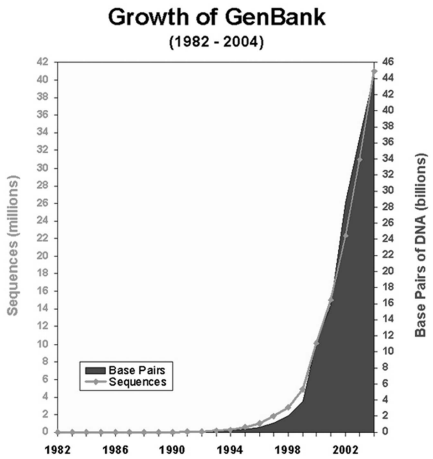

Fig. 2. Trend in sequence database growth (from http://www.ncbi.nlm.nih.gov/Genbank/genbankstats.html).

reporting phase, which can result in inverted scaling (i.e., using more database fragments and more processors gives a longer runtime, rather than a shorter runtime) [22]. We also extensively profiled MPI-BLAST and verified the presence of these performance issues on a distributed-memory architecture (Linux cluster) as well. Profiling of MPI-BLAST on MPP2, a 11.8 TFLOP Linux cluster installed at PNNL, has revealed that the gains from parallelization by fragmenting the database are more than offset by increased time of master node processing (which is serial) when the process count and, hence, the number of database fragments, is increased beyond 8. Fig. 1 illustrates the poor scaling we observed using MPI-BLAST when using additional processors while correspondingly increasing the number of database fragments.

This poor scaling is partly due to the fact that MPP2 has more than adequate memory for the entire *nr* database to fit on each processor, so the superlinear speedup due to removing the paging bottleneck reported in [21] is not observed. Second, the processing speed of our architecture is such that the communication time, printing time, and spin-waiting associated with the collating phase outweighs the local query time when only a few processors are used. This serial phase takes a proportionally higher amount of the total runtime when more processors are added, preventing scaling beyond a few processors. Other efforts have used similar database partitioning schemes [23]. It was this scalability problem associated with database partitioning that led to the development of ScalaBLAST.

Orthogonally, along with the service scheduling previously discussed, software schemes have been developed to rapidly calculate alignments by scheduling queries separately [24], [25], [26], [27], [28]. This approach has the advantage that, in most applications, separate queries are unrelated and therefore can be dispatched concurrently. However, the increasing size of target databases will increasingly deteriorate the performance of these approaches as repeated file access becomes more frequent during alignment calculations. Hybrid methods also exist that combine database partitioning and query scheduling [29].

One key bottleneck in sequence alignment efficiency is related to I/O. Piers is an application which attempts to mitigate this performance bottleneck by using a hash table structure to parse the target database, eliminating the need for each sequence to be compared to the entire database [30]. This makes it possible in many cases to leave a smaller data structure in core, obviating the need for repeated file access during a run. However, Piers relies on a heuristic to parse the database and does not give identical results to BLAST. PioBLAST optimizes I/O by employing MPI-IO to increase the efficiency of accessing the database fragments required by MPI-BLAST [22].

3 Motivation and Design Goals

3.1 Databases Are Growing Faster than Memory

As public gene and protein sequence databases continue to increase in size at an exponential rate, the need to deal with very large database files becomes more urgent. Fig. 2 illustrates the growth of one public sequence repository, GenBank. The largest of the protein databases, the nonredundant protein database, is currently about 2GB. Many computers can easily fit files of this size entirely in-core. However, we expect that the size of this database will continue to grow exponentially as the rate of high-throughput sequencing technology continues to increase and the number of organisms being sequenced continues to increase.

Even though, today, most databases can fit in-core in a single enclosure, ScalaBLAST shares only one copy of any target database between all process groups. Benchmark results shown in this paper therefore contain representative communication overhead similar to what one would expect in the future when sharing a *single* large database image is required.

In response to this explosion in database size, the current implementation of BLAST contains a formatting routine that converts ASCII text FASTA files containing sequence information to a collection of binary files to ease the space limitations associated with large sequence databases. Database formatting automatically splits databases that exceed two billion letters into separate "volumes," ensuring that 4-byte integers can be used to index any database volume. One consequence of this is that large databases cannot be completely stored in-core, even when a large shared memory interface is present. ScalaBLAST was designed to get around this problem by enabling a cluster having sufficient aggregate memory to keep an entire large

database in-core by sharing a single copy of the database between process groups.

3.2 Scalability Is a Consequence of Independent Queries

Query scheduling methods capitalize on the independent nature of separate queries, but are all limited by the fact the target databases tend to be very large and are growing more quickly than single processor memory. As the anticipated growth rate of publicly available databases continues to be essentially exponential, a new approach to parallelism is required. On the other hand, achieving parallelism by *distributing a single query* versus scheduling queries has advantages and disadvantages [31]. Algorithms that distribute the task of scoring a single query generally suffer because of the communication required or complex data dependency to coalesce the partial results into a single result for a given query. But, distributing a single scoring task generally reduces the per-processor memory requirement. ScalaBLAST was implemented with a query scheduling algorithm to achieve scalability indicative of sequence scheduling schemes. Combined with database sharing, ScalaBLAST scales well *and* is able to accommodate very large databases, a combination which has proven elusive in high-performance sequence analysis.

4 Technical Approach

ScalaBLAST uses a target database that is physically distributed but can be still accessed in a "shared memory" style at runtime. This eliminates the need to preprocess each possible target database into fragments, a step that is required when using MPI-BLAST and which must be incrementally repeated each time the database is updated. Database *sharing* (rather than prepartitioning) ensures that, even though each process may only *handle* half of a single query, they all *see* the same entire database. So, the calculation of E-values (a measure of the likelihood of sequences appearing homologous by chance) is straightforward using ScalaBLAST. By comparison, calculation of the E-value was problematic with early versions of MPI-BLAST [21] and still requires special processing. Database sharing also makes it possible to fit extremely large databases into local memory even with moderate per-processor memory. The Global Array interface allows this "shared memory" approach to be used on distributed memory or true shared memory architectures, increasing portability and providing the functionality to hide memory latency by overlapping communication with computations.

We achieve the first layer of parallelism by dividing single queries over a few processors (forming an MPI process group) and allowing that process group to merge the partial results and print into its own output file. This ensures that using more processors on a query list does not result in the need for cumbersome merging over many small partial results. Further parallelism is achieved by dividing the query list itself over the available process groups. This layered parallelism allowed ScalaBLAST to achieve near-linear scaling in proportion to the number of queries being requested. For 1,000 queries, we observed excellent scaling to 128 processors. For whole genome or whole proteome comparisons (involving 10,000 queries or more), we have observed linear scaling to thousands of processors. We will now elaborate on key elements of the overall parallelization strategy.

4.1 Distributing the Database Using Global Arrays

Rather than force the user to preformat databases depending on the target database and the number of processors to be used in the query (a required step with MPI-BLAST), we developed a method to read the target database files at runtime and pack their contents into several shared arrays. There are two key arrays used for this. The first array contains the actual sequence information. On our test platforms (little endian), this must be byte-swapped from the database files at runtime. Normally, this byte-swapping is done at the time the information is read from the files and repeated again each time an entry is read. We byte-swap the entire sequence database as it is read from the file, eliminating the need for repeated byte-swapping of the same sequences. Sequences are packed sequentially into a sequence array that is visible to all the process groups. A second "index" array is used to store information about the starting location of each sequence. For example, the first and second entries of this index array contain the starting location of the first and second sequences in the sequence array, which can be used to find the starting and ending point of the first sequence in the sequence array. To complete a sequence lookup, we have modified the standard BLAST database reading function so that, rather than opening the sequence file, the sequence is simply looked up in the proper array object.

4.2 Breaking Up a Query List over Process Groups

To exploit available parallelism effectively, ScalaBLAST relies on multilevel parallelism based on processor groups. Each process group is composed of a few processors (two on MPP2, which is based on two-way SMP nodes). The idea is to have processors forming a group to be a part of the same SMP node, which enables very fast interprocessor communication due to the internal use of shared memory for allocating global arrays in GA [32], [33].

An initial query list is divided between process groups using a static load balancing scheme. Each process group receives a new file containing, as closely as possible, the same amount of "work" units. Total work is defined as the number of characters in the combined sequences plus an overhead of 225 characters per each new sequence. For example, if the initial query list contains 100 sequences to be divided between four processors (two process groups), one may expect that each group should get around 50 sequences. However, if the first 50 sequences all contain only 100 characters each and the last 50 sequences contain 1,000 characters each, this would result in a huge load balance problem. So, using this simple algorithm, the total work to be done is 100*50 + 1,000*50 + 225*100 units or 77,500 units. Divided between the two process groups, both groups would get as close to 38,750 units as possible. The first half of the list contains only 100*50 + 225*50 units or 16,250 units. So, the first process group will get all the first 50 sequences, plus some of the remaining sequences. If the first process group receives an additional 18 of the longer

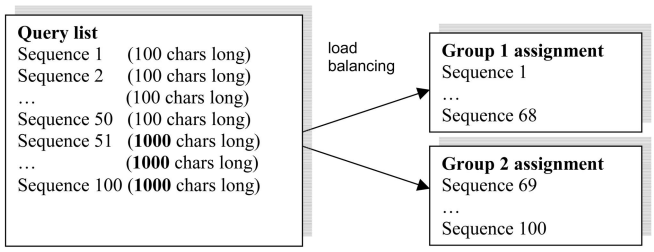

Fig. 3. Schematic of load balancing used by ScalaBLAST. Load balancing is achieved by assigning each process group roughly the same number of query characters.

sequences, the total work will be 16,250 + 18*225 + 18*1,000 = 38,300 units, which is as close as possible to the target of 38,750. So, the first process group will get the first 68 sequences in the query list and the remaining 32 sequences will be processed by the second process group. The sequence overhead of 225 was derived empirically from trial-and-error using many different query sizes and compositions. Once the load balance has been calculated, each process group creates a new input file based on its assignments. Fig. 3 illustrates the process group assignments for this sample case.

4.3 Process Group Tasks: The Query Phase

Once the process groups have created their local query lists, sequence alignment calculation of those lists against the target database proceeds. When the database is initially read into the sequence array, the "midpoint" residue is located. That is, the character that is in the exact center of the sequence list and its corresponding sequence are recorded.

In each process group, process 0 performs an alignment calculation of the current query against all the sequences from the first to the center sequence and process 1 performs an alignment calculation of the same query against all the remaining sequences in the sequence array. Each process posts the partial results thus obtained to a "results" array object that is only visible to the process group. Additional bookkeeping information is also kept to be used during the result reporting phase. The two processes can perform their partial alignments independently of each other, so no synchronization is required until it is time to dump a set of results to output files. Currently, this query phase is iterated over the first 20 queries in the local list and all the partial results are stored before dumping output occurs.

4.4 Process Group Tasks: Reporting Results

After the query phase is complete, the processes independently move into the reporting phase. Because partial results are posted, it is irrelevant which process actually does the reporting for a query. It is only necessary to ensure that, for a given query, all processes in the group have completed their alignment calculation and posted the partial results. Because the query phase is iterated over several queries it is rare that a process needs to block until these results are posted. Process 0 of the group prints results from odd numbered queries, and process 1 of the group prints results from even numbered queries. To print, process 1 retrieves the locations of the two partial results for the first query in the "results" array. These partial results are merged into a single results list and dispatched to the normal BLAST reporting routine. Process 1 then retrieves the locations of the two partial results for the third query in the "results" array, then merges and dispatches these results for reporting. Process 1 proceeds until all the odd numbered queries have been dumped to the output file. Process 0 follows a similar course for dumping the even numbered queries. Both processes write to output files unique from each other and unique from all the other output files of other process groups. In this way, I/O is achieved in the most highly parallel way possible and the need for synchrony between processors with respect to printing and alignment calculations is removed. If any queries remain after reporting, the algorithm returns to the query phase.

ScalaBLAST distributes the reporting tasks among the process groups so that each processor writes its results to a unique file. This is important in terms of performance because it allows I/O to occur in parallel, rather than force processors to wait for a master process to complete writing tasks. Each process group is responsible for printing its own results and can proceed with the next tasks independently of other process groups. Additionally, a query containing 10,000 sequences is expected to produce almost 5 GB of output. A single text file of this size could be created after the run by concatenating the results of the original output. But, dealing with files of this size can sometimes be cumbersome or even impossible. For many users, it may in fact be easier to deal with a number of smaller files than a single monolithic output file.

4.5 Prefetching Sequence Blocks to Hide Latency on Distributed Memory Systems

Comparing to the specialized vector supercomputers that provide very high interconnect bandwidth [34], most commodity clusters offer less bandwidth relative to the CPU performance (bytes/op ratio). However, modern interconnect such as the Quadrics Elan-4 deploy intelligent network interfaces (NICs) that mitigate some of the bandwidth shortages by offloading most of the communication costs to the NIC and supporting nonblocking communication. This in turn allows some applications to hide communication costs by overlapping communication with computations.

In our database sharing approach, illustrated in Fig. 4, the target sequence database is evenly distributed over the memory of processors being used at runtime. But, sharing the database in distributed memory architectures presents a challenge in terms of managing memory access. Each sequence in the database is serially scored against the query sequence for the first pass over the database. The brute force implementation of database sharing would require blocking memory fetch operations to be performed on each sequence in the database for each process group per query. This creates a memory-access bottleneck that is exacerbated in distributed memory architectures. A further difficulty is that, when this approach is scaled to higher processor counts, an increasing amount of the target database is in nonlocal memory segments, increasing the fraction of nonlocal blocking *get* operations. ScalaBLAST

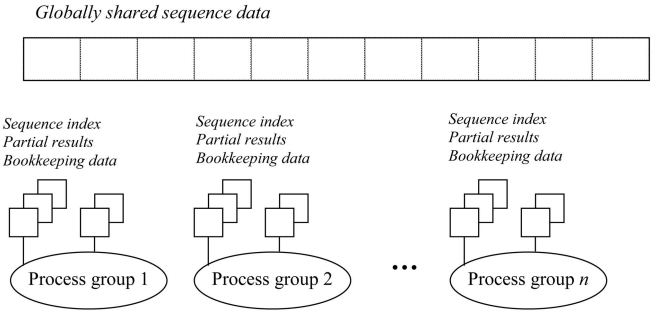

Fig. 4. Scope and type of data stored in global arrays defined on process groups. Each group has its own local copy of the sequence index array, partial results arrays, and all the bookkeeping arrays. The process groups share a single copy of the sequence data array.

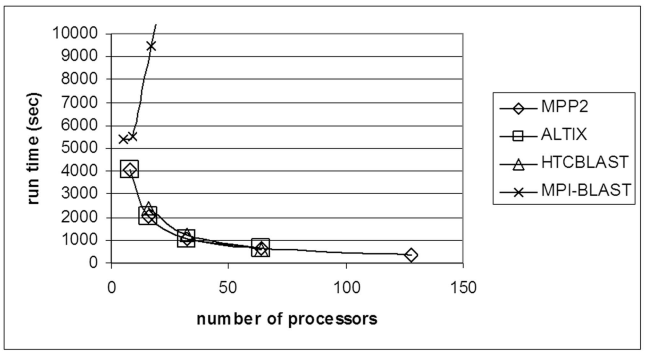

Fig. 5. Wall clock times of ScalaBLAST on distributed memory architecture (MPP2) and SGI Altix (ALTIX) compared to HTC-BLAST on the SGI Altix and MPI-BLAST 1.3.0 on MPP2 for 1,000 queries (709kB of input). For 32 worker processors, MPI-BLAST had a wall-clock time of 17,398 seconds.

uses a nonblocking prefetch algorithm that makes it possible for remote data access to large blocks to occur while other meaningful calculations are being done locally. The implementation relies on nonblocking *get* operations available through the Global Array toolkit to hide memory latency on distributed memory systems by using sequence block prefetching [35]. When a processor attempts to retrieve a sequence from the database, first the local prefetch buffer is checked for the presence of the correct sequence. If the sequence is not entirely contained in the current local buffer, a wait call is issued to ensure that the previous nonblocking *get* has completed, and the current local buffer is updated to point to the new memory segment. An additional nonblocking *get* is initiated for the segment that will be required when the last sequence in the current buffer is reached. In this way, one nonblocking *get* is always outstanding (and, therefore, in progress) until the end of the database is reached to make sure that sequence segments are available in local memory when they are needed.

5 Experimental Results

ScalaBLAST was benchmarked on two systems representing shared as well as distributed memory architectures using the nonredundant protein database (*nr*) containing 1,541,362 sequences (503,870,249 characters) or was normalized to this size if the database was larger at the time of the benchmark. The nr database is maintained by NCBI to contain a nonredundant record of all proteins registered to date. Hence, one can query against this database to find sequence homology and alignments against the genome of any species which has been sequenced so far. Queries were performed using the FASTA files containing 1,000 sequences, with a total input size of 709 kB.

MPP2 is a distributed memory system composed of 1,960 Intel Itanium II processors running the Linux OS connected with a high-performance Quadrics Elan-4 interconnect. Database, input, and output files were located on a globally-mounted high-performance Lustre filesystem, optimized in the MPP2 environment for parallel I/O. MPP2 contains so-called "fat" nodes with 8 GB per node and "thin" nodes with 6 GB per node. Each SMP node deploys two CPUs running at 1.5GHz clock speed. The SGI Altix at PNNL is a shared memory architecture composed of 128 1.5GHz Itanium II processors running the Linux 2.4.20. The total memory of the system is 256 GB. Database, input, and output files were located in the local filesystem of the Altix.

The experimental evaluation of ScalaBLAST was performed on the Linux cluster and the SGI Altix. On the cluster, we also installed MPI-BLAST 1.2.1 and the latest MPI-BLAST 1.3.0 for comparison. The performance evaluation of MPI-BLAST on the very similar configuration of the SGI Altix was reported elsewhere [32]. In addition, we tested the SGI HTC-BLAST on the Altix. The reported wall clock time corresponds to all the elements of the application, including the initial setup activities such as byte swapping for the input database that is executed sequentially.

Similarly to results reported by others [32] on both platforms, we observed inverted scaling for MPI-BLAST (i.e., longer runtimes when using more processors) when running on more than eight processors, a problem that we observed as well and which is illustrated in Fig. 1. The best runtime for MPI-BLAST against 1,000 queries on MPP2 was 5,397 seconds and was achieved using four worker processors (five processors, total) after dividing the nr database into four parts using MPI formatdb (included with the MPI-BLAST distribution). Using the sequential BLAST application running on one processor, this query required 33,322 seconds or 9.3 hours.

Based on results reported in Fig. 5, it is clear that ScalaBLAST scales well to 128 processors. Its runtimes are significantly faster than the best time we observed for MPI-BLAST. ScalaBLAST running on the ALTIX and MPP2 also had slightly faster runtimes than HTC-BLAST for each processor count.

ScalaBLAST has significant advantages over HTC-BLAST in that ScalaBLAST is portable to other architectures where a much higher processor count is possible, enabling scaling to process counts far beyond that available through conventional shared memory architectures. Perhaps more importantly, ScalaBLAST does not rely on hardware shared memory, but rather takes advantage of software implementation of shared memory available through Global Arrays.

Scaling illustrated in Fig. 5 is achieved through efficient sharing of the target database among process groups and by eliminating the output bottleneck by using a separate

TABLE 1
Fraction of ScalaBLAST Runtime Spent in Setup, Query, and Writing Tasks

	Setup (%)	Query (%)	Writing (%)
100 queries 8 processors	2.6±0.7	94.9±0.6	2.59±0.6
1000 queries 8 processors	0.056±0.0003	98.5±0.8	1.4±0.8
1000 queries 32 processors	0.223±0.005	98.3±0.8	1.5±0.8

output file for each processor. Table 1 shows the fraction of time spent in setup, the query phase, and the writing phase for three representative runs. Note that increasing either the job size or the number of processors increases the fraction of the runtime ScalaBLAST dedicated to performing the actual query. Hence, the writing bottleneck associated with collating and writing results from a single I/O node is bypassed. Even for the small 100 query jobs, for which the total setup time is a greater fraction of the total run time, greater than 90 percent of ScalaBLAST execution time is spent in the query phase.

Fig. 6 illustrates a comparison of scaling of ScalaBLAST with and without prefetching for the same database size and queries on up to 128 processors. Prefetching is deployed to hide communication latency on clusters. The corresponding difference in runtime demonstrates the performance benefit that is gained by judiciously using nonblocking GA one-sided interfaces that were optimized to enable overlapping communication with computations [35]. The cost of sending sequence information using blocking calls is significant in a distributed memory environment, so there is a substantial benefit from using prefetching on our Linux cluster. The runtimes of ScalaBLAST using prefetching on MPP2 were nearly identical to those on the ALTIX, as illustrated in Fig. 5.

To demonstrate scaling of ScalaBLAST on a much larger test set, we used a collection of protein domain family alignments used to characterize family membership (PFAM) [36]. This collection of approximately 448,000 protein sequences was queried against the nr database on MPP2 using a varying number of processors. Each run was allowed to proceed for 195 minutes, processing as much of the list as possible. For the 1,500 processor run, 445,989 of the sequences were processed in 195 minutes. The work factor was calculated for each run to assess the throughput of sequence processing per processor. For each run, the work factor is equal to the number of queries completed per processor per minute per 1 million sequences in the database. The results of this benchmark are illustrated in Fig. 7. Work factor is normalized to a database size of 1 million sequences for future comparison between nr databases of differing size. Using nr databases taken at different times, we found that query time is very nearly linear with respect to number of sequences in the database (results not shown).

Fig. 7 illustrates that, when using more than one node (two processors on MPP2), the amount of work done by ScalaBLAST per processor does not decrease when using more processors. This is analogous to ideal scaling as indicated in Fig. 7 by the solid line, even when running on 1,500 processors.

6 Application Experience Using ScalaBLAST

ScalaBLAST has already demonstrated its applicability and effectiveness as a valuable tool for bioinformatics research. For example, it was used to perform whole proteome searches against the updated nr database (2,440,549 sequences). This set of queries contained 48,690 sequences (17.6 MB of input).

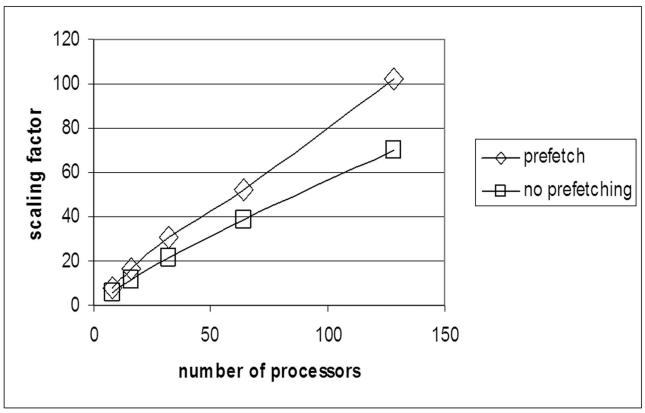

Fig. 6. Prefetching sequence blocks using nonblocking get operation hides memory access latency on distributed memory architecture (MPP2). Without prefetching, runtimes are approximately 1.5 times slower on MPP2 than when prefetching is used. Scaling factor is calculated with respect to total runtime of serial standalone BLAST of 33,322 seconds.

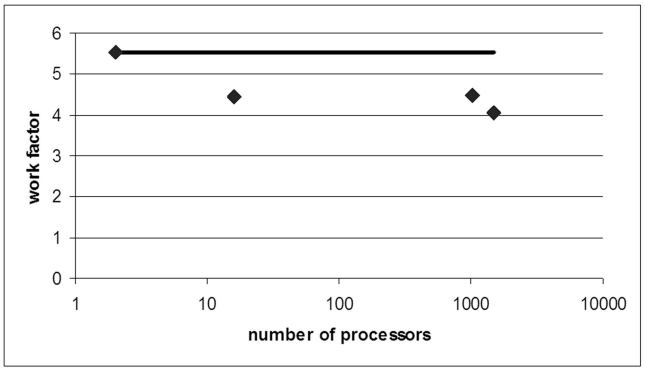

Fig. 7. Work factor indicates the throughput of queries handled by each processor. The work factor is calculated as the number of queries performed per processor per minute per million sequences in the target database. The solid line shows ideal work factor that would be achieved for a perfectly scaling application. Note that, for more than two processors (i.e., more than one node), the work factor is *essentially constant* with respect to number of processors.

Using ScalaBLAST on the SGI Altix, this run completed in 9.4 hours using 64 processors. The results from this run will be used to perform genome context data mining. This data-intensive scientific field requires the output of entire proteome searches to be used as input for using contextual information (the locality of a sequence with respect to neighbor sequences) to infer evolutionary relationships between sequences. Generating the enormous data required to perform genome context analysis has traditionally been a prohibitive bottleneck to the overall workflow. In this case, ScalaBLAST is being used to dramatically speed up the rate of generating input for this analysis by exploiting massively parallel systems with large globally addressable memory.

7 Discussion

Experimental results demonstrate that ScalaBLAST performs as well as the vendor optimized proprietary HTC-BLAST shared-memory implementation and significantly better than MPI-BLAST for real-world queries against the nonredundant protein database. The advantages of ScalaBLAST are that it scales well *and* is portable to both shared memory and distributed memory architectures. ScalaBLAST gives true high-performance on the high-end systems we tested, in terms of time-to-solution and scaling, representing a significant improvement in availability over HTC-BLAST with virtually equivalent run times.

Portability and the capacity to operate on extremely large databases are conferred by use of Global Arrays, which hides architectural details by allowing for logical "shared memory access" on any platform. Global Arrays also are instrumental in distributing the database evenly over the processor-attached memory segments in the shared architecture to prevent memory-controller bottlenecks. In fact, we observed virtually no difference between the runtime of ScalaBLAST on the ALTIX and on MPP2 when sequence block prefetching was used to hide memory access latency on MPP2. Prefetching was performed based on nonblocking get operations available with the Global Array toolkit. This approach makes it possible for sequence analysis tools to operate efficiently in both shared memory and distributed memory architectures. It enables queries against extremely large target databases for large data-intensive sequence analysis applications, a combination that is likely to become more important as the growth of publicly available databases continues to outpace increases in per-processor memory availability and as the push toward data-intensive applications matures.

Though the current size of these public databases is small enough to fit in-core on many architectures (including our SGI Altix and MPP2), we demonstrate proof-of-concept that database sharing can be done efficiently using a software shared memory interface—Global Arrays. For all benchmark results reported, a *single copy of the target database* was shared by all process groups, thereby incurring a communication overhead proportional to what we would expect from larger databases that do not fit in-core on a single enclosure. Even with this penalty (i.e., communication penalty for distributed architecture and corresponding memory controller penalty on shared memory), ScalaBLAST scales well in both shared memory and distributed memory architectures and effectively hides the latency associated with accessing sequence data from the shared databases.

We see two important trends that are likely to continue in the foreseeable future converging in the field of bioinformatics: 1) exponential growth of the size of publicly available sequence databases and 2) a push for larger scale informatics driven science to be performed using sequence analysis on these databases. ScalaBLAST was developed as a solution to these challenges that can be broadly applied to sequence analysis in general. The methodology employed by ScalaBLAST can be used, in principle, for any sequence analysis tasks that can be performed by searching sequentially through a very large database. ScalaBLAST lays the foundation of managing extremely large databases—even ones that are too big to fit in memory for any single machine. Using ScalaBLAST as a test application, this database-sharing scheme has been shown to scale well for real-world problems. By hiding memory latency in ScalaBLAST, we were able to show that our database sharing strategy is practical for use on both shared memory and distributed memory architectures to achieve excellent scalability from whatever resources are available to the researcher. We chose BLAST as the test application to demonstrate proof-of-concept that global array-enabled database sharing works well on both distributed memory and shared memory architectures. Because sequence alignment tools in general operate on large sequence databases, the concepts we developed in ScalaBLAST are widely applicable to many other bioinformatics tools.

An externally accessible ScalaBLAST server will be set up allowing users to submit large queries to a test cluster. This will provide a vehicle for community access to ScalaBLAST for testing and to identify areas for further development.

8 Conclusions

With the development of ScalaBLAST, we have demonstrated that the combination of software-enabled database sharing and query scheduling without the use of specialized hardware solves scalability problems and memory limitations: the two main shortcomings of existing high-performance sequence alignment applications. ScalaBLAST scales linearly to thousands of processors on distributed memory architecture and to machine capacity on shared memory architectures for whole genome-sized sequence alignment tasks, which are driving the growing area of informatics-driven science. ScalaBLAST also can accommodate very large databases in-core, an essential feature for achieving high machine utilization as sequences database sizes continue to grow more quickly than per-processor core memory. As an openly accessible tool, ScalaBLAST will vastly improve time-to-solution for large sequence alignment searches, enabling "tree of life" analysis to be performed *rapidly* on newly sequenced organisms and also on organisms, like human, for which whole genome sequence alignment is currently intractable.

Acknowledgments

The research described in this paper was supported in part by the US Department of Energy, Office of Advanced

Scientific Computing Research through the "Exploratory Data Intensive Computing for Complex Biological Systems" project and through the LDRD Program at the Pacific Northwest National Laboratory, a multiprogram national laboratory operated by Battelle for the US Department of Energy under Contract DE-AC06-76RL01830. This research was performed in part using the Molecular Science Computing Facility (MSCF) in the William R. Wiley Environmental Molecular Sciences Laboratory, a national scientific user facility sponsored by the US Department of Energy's Office of Biological and Environmental Research and located at the Pacific Northwest National Laboratory. Pacific Northwest is operated for the US Department of Energy by Battelle. The authors would also like to thank Doug Baxter for conversations and helpful input during the course of this project.

REFERENCES

[1] T. Smith and M. Waterman, "Overlapping Genes and Information Theory," *J. Theoretical Biology*, vol. 91, pp. 379-380, 1981.
[2] T. Smith, M. Waterman, and W. Fitch, "Comparative Biosequence Metrics," *J. Molecular Evolution*, vol. 18, pp. 38-46, 1981.
[3] S. Needleman and C. Wunsch, "A General Method Applicable to the Search for Similarities in the Amino Acid Sequence of Two Proteins," *J. Molecular Biology*, vol. 48, pp. 443-453, 1970.
[4] S. Altschul, W. Gish, W. Miller, E. Myers, and D. Lipman, "Basic Local Alignment Search Tool," *J. Molecular Biology*, vol. 215, pp. 403-410, 1990.
[5] S. Altschul, T. Madden, A. Schaffer, J. Zhang, Z. Zhang, W. Miller, and D. Lipman, "Gapped BLAST and PSI-BLAST: A New Generation of Protein Database Search Programs," *Nucleic Acids Research*, vol. 25, pp. 3389-3402, 1997.
[6] S. Brenner, C. Chothia, and T.J.P. Hubbard, "Assessing Sequence Comparison Methods with Reliable Structurally Identified Distant Evolutionary Relationships," *Proc. Nat'l Academy of Science US*, vol. 95, pp. 6073-6078, 1998.
[7] B. Webb, J. Liu, and C. Lawrence, "BALSA: Bayesian Algorithm for Local Sequence Alignment," *Nucleic Acids Research*, vol. 30, pp. 1268-1277, 2002.
[8] A. Krause, J. Stoye, and M. Vingron, "Large Scale Hierarchical Clustering of Protein Sequences," *BMC Bioinformatics*, vol. 6, p. 15, 2005.
[9] H. Sofia, G. Chen, B. Hetzler, J. Reyes-Spindola, and N. Miller, "Radical SAM, A Novel Protein Superfamily Linking Unresolved Steps in Familiar Biosynthetic Pathways with Radical Mechanisms: Functional Characterization Using New Analysis and Information Visualization Methods," *Nucleic Acids Research*, vol. 29, pp. 1097-1106, 2001.
[10] A. Darling, L. Carey, and W.-C. Feng, "The Design, Implementation, and Evaluation of mpiBLAST," *Proc. ClusterWorld*, 2003.
[11] N. Camp, H. Cofer, and R. Gomperts, "High-Throughput BLAST," 1998.
[12] J. Nieplocha, R. Harrison, and R. Littlefield, "Global Arrays: A Nonuniform Memory Access Programming Model for High-Performance Computers," *J. Supercomputing*, vol. 10, pp. 197-220, 1996.
[13] X. Meng and V. Chaudhary, "Bio-Sequence Analysis with Cradle's 3SoCTM Software Scalable System on Chip," *Proc. ACM Symp. Applied Computing*, 2004.
[14] K. Muriki, K. Underwood, and R. Sass, "RC-BLAST: Towards a Portable, Cost-Effective Open Source Hardware Implementation," *Proc. HICOMB 2005, Fourth IEEE Int'l Workshop High-Performance Computational Biology*, 2005.
[15] J. Wang and Q. Mu, "Soap-HT-BLAST: High-Throughput BLAST Based on Web Services," *Bioinformatics*, vol. 19, pp. 1863-1864, 2003.
[16] R. Bjornson, A. Sherman, S. Weston, N. Willard, and J. Wing, "TurboBLAST(r): A Parallel Implementation of BLAST Built on the TurboHub," *Proc. 16th Int'l Parallel and Distributed Processing Symp. (IPDPS)*, 2002.
[17] T. Braun, T. Scheetz, G. Webster, A. Clark, E. Stone, V. Sheffield, and T. Casavant, "Identifying Candidate Disease Genes with High-Performance Computing," *J. Supercomputing*, vol. 26, pp. 7-24, 2003.
[18] "Cluster Computing: SGI Altix Screams on Itanium," *HPC Wire*, vol. 12, 2003.
[19] H. Nicholas, G. Giras, V. Hartonas-Garmhausen, M. Kopko, C. Maher, and A. Ropelewski, "Distributing the Comparison of DNA and Protein Sequences across Heterogeneous Supercomputers," *Proc. ACM/IEEE Conf. Supercomputing*, 1991.
[20] T. Rognes, "ParAlign: A Parallel Sequence Alignment Algorithm for Rapid and Sensitive Database Searches," *Nucleic Acids Research*, vol. 29, pp. 1647-1652, 2001.
[21] M. Salisbury, "Parallel Blast: Chopping the Database," *Genome Technology*, pp. 21-22, 2005.
[22] H. Lin, X. Ma, P. Chandramohan, A. Geist, and N. Samatova, "Efficient Data Access for Parallel BLAST," *Proc. 19th Int'l Parallel and Distributed Processing Symp. (IPDPS)*, 2005.
[23] D. Mathog, "Parallel BLAST on Split Databases," *Bioinformatics*, vol. 19, pp. 1865-1866, 2003.
[24] K. Hokamp, D. Shields, K. Wolfe, and D. Caffrey, "Wrapping Up BLAST and Other Applications for Use on UNIX Clusters," *Bioinformatics*, vol. 19, pp. 441-442, 2003.
[25] W. Gish, 1996-2004.
[26] J. Grant, R.J. Dunbrack, F. Manion, and M. Ochs, "BeoBLAST: Distributed BLAST and PSI-BLAST on a Beowulf Cluster," *Bioinformatics*, vol. 18, pp. 765-766, 2002.
[27] M. Schmollinger, K. Nieselt, M. Kaufmann, and B. Morgenstern, "DIALIGN P: Fast Pair-Wise and Multiple Sequence Alignment Using Parallel Processors," *BMC Bioinformatics*, vol. 5, p. 128, 2004.
[28] C. Wang and E. Lefkowitz, "SS-Wrapper: A Package of Wrapper Applications for Similarity Searches on Linux Clusters," *BMC Bioinformatics*, vol. 5, p. 171, 2004.
[29] R. Braun, K. Pedretti, T. Casavant, T. Scheetz, C. Birkett, and C. Roberts, "Parallelization of Local BLAST Service on Workstation Clusters," *Future Generation Computer Systems*, vol. 17, 2001.
[30] X. Cao, S.C. Li, B.C. Ooi, and A.K.H. Tung, "Piers: An Efficient Model for Similarity Search in DNA Sequence Databases," *SIGMOD Record*, vol. 33, pp. 39-44, 2004.
[31] R. Costa and S. Lifschitz, "Database Allocation Strategies for Parallel BLAST Evaluation on Clusters," *Distributed and Parallel Databases*, vol. 13, pp. 99-127, 2003.
[32] J. Nieplocha, J. Ju, and T. Straatsma, "A Multiprotocol Communication Support for the Global Address Space Programming Model on the IBM SP," *Proc. Euro-Par*, 2000.
[33] J. Nieplocha, M. Krishnan, B. Palmer, V. Tipparaju, and Y. Zhang, "Exploiting Processor Groups to Extend Scalability of the GA Shared Memory Programming Model," *Proc. ACM SIGMicro Computing Frontiers*, 2005.
[34] L. Oliker, A. Canning, J. Carter, J. Shalf, and S. Ethier, "Scientific Computations on Modern Parallel Vector System," *Proc. ACM/IEEE SuperComputing Conf. '04*, 2004.
[35] J. Nieplocha, B. Palmer, V. Tipparaju, M. Krishnan, H. Trease, and E. Apra, "Advances, Applications, and Performance of the Global Arrays Shared Memory Programming Toolkit," *Int'l J. High-Performance Computing Applications*, vol. 20, pp. 203-231, Summer 2006.
[36] A. Bateman, L. Coin, R. Durbin, R.D. Finn, V. Hollich, S. Griffiths-Jones, A. Khanna, M. Marshall, S. Moxon, E.L. Sonnhammer, D.J. Studholme, C. Yeats, and E. SR., "The PFam Protein Families Database," *Nucleic Acids Research*, vol. 32, pp. D138-D141, 2004.

Christopher Oehmen received the BA degree in physics and mathematics from Saint Louis University in 1995 as a four-year presidential scholar. He subsequently received the MS and PhD degrees in biomedical engineering from the Joint Graduate Program in Biomedical Engineering, The University of Memphis and The University of Tennessee, Memphis Health Science Center in 1999 and 2003, respectively. During his graduate studies, he studied syncitial pacing in mammalian cardiac pacemaker cell systems using high-performance computational modeling. During his PhD studies, he was supported by the US Department of Energy Computational Science Graduate Fellowship (Krell institute). He is currently a member of the Computational Biology and Bioinformatics Group in the Computational Sciences and Mathematics Division at Pacific Northwest National Laboratory, where he is involved in development of high-performance codes for applications in bioinformatics and machine learning applied to biological systems. He is a member of the IEEE and the IEEE Computer Society.

Jarek Nieplocha is a laboratory fellow and the technical group leader of the Applied Computer Science Group in the Computational Sciences and Mathematics Division of the Fundamental Science Division at Pacific Northwest National Laboratory (PNNL). He is also the chief scientist for high-performance computing in the Computational Sciences and Mathematics Division. His area of research has been in optimizing performance of collective and one-sided communication operations on modern networks, runtime systems, parallel I/O, and scalable programming models. He received four best paper awards at conferences in high-performance computing: IPDPS '03, Supercomputing '98, IEEE High-Performance Distributed Computing HPDC-5, and IEEE Cluster '03 conference, and an R&D-100 award. He has authored and coauthored more than 80 peer reviewed papers. Dr. Nieplocha participated in the MPI Forum in defining the MP-2 standard. He is also a member of the editorial board of the *International Journal of Computational Science and Engineering*. He is a member of the IEEE Computer Society.