Processor and Data Scheduling for Online Parallel Sequence Database Servers

Heshan Lin* Xiaosong Ma*† Jiangtian Li* Ting Yu* Nagiza Samatova*†

Department of Computer Science, North Carolina State University*
Computer Science and Mathematic Division, Oak Ridge National Laborato†
hlin2@ncsu.edu ma@cs.ncsu.edu jli3@ncsu.edu yu@csc.ncsu.edu samatovan@ornl.gov

ABSTRACT

Scientific databases often possess data models and query workload quite different from commercial ones and are much less studied. We look into one such instance by studying high-throughput query processing on biological sequence databases, a fundamental task performed daily by millions of scientists. Here the databases are stored in large annotated files and each query is an expensive dynamic programming task based on a full-database scan. As both the database size and search complexity call for parallel/distributed processing, web-based online parallel searches have become popular. By efficiently utilizing and sharing high-end computing resources while keeping the interactiveness and sequential interface of query processing, it is an ideal choice for research institutes and companies.

Our research presented in this paper indicates that intelligent resource allocation and scheduling are crucial in improving the overall performance of a parallel sequence search database server. Failure to consider either the parallel computation scalability or the data locality issues can significantly hurt the system throughput and query response time. In addition, no single strategy works best in all circumstances. In response, we present several dynamic scheduling techniques that automatically adapt to the search workload and system configuration in making scheduling decisions. Evaluation results using a simulator (which is verified against real-cluster experiments) show the combination of these techniques delivers up to an order-of-magnitude performance improvement across various workload and system parameter combinations.

Keywords

Biological sequence databases, parallel query processing, cluster web servers, data placement, task scheduling

1. INTRODUCTION

Databases in the scientific computing community possess a combination of unique characteristics. First, they are often implemented as large flat files. Second, queries are both computation- and data-intensive, performing non-trivial algorithms over large amounts of data. Therefore, scientific database processing, especially when carried out on parallel or distributed platforms, requires careful examination of the intertwined computation and data management issues.

In this paper, we address this problem in the biology field, where fundamental sequence database search tasks such as BLAST [1] are performed routinely by scientists. Given a query sequence (typically a newly identified sequence), the BLAST family tools search through a database of known sequences and returns sequences that are similar to the query sequence. This process is very useful in determining the function of new sequences, as well as in phylogenetic profiling and bacterial genome annotation.

The growing database size as a result of the growing speed of sequence discovery, however, has made the search job increasingly expensive. Being both computation-intensive and data-intensive, sequence search tasks are becoming overwhelming for a single desktop workstation. For example, the NT database hosted at NCBI (National Center for Biological Information) currently is sized at 5.6 GB when formatted for search. As the size of the sequence databases and the speed of generating new sequences through experiments both grow, sequence database search tasks easily outgrow the processing and storage capacity of a single workstation.

Designed to expedite sequence search query processing, parallel BLAST tools have been built and become increasingly popular (more details about them will be given in Section 2). But such tools require to be used in a cluster environment through parallel job submission and execution interfaces, which involves large operational overhead. In contrast, building cluster web servers that provide transparent parallel BLAST search services has become a desirable solution. It enables both seamless resource sharing and friendly interface: users submit queries through a front-end node, without dealing with the hassle of parallel execution. This approach has become popular with institutes such as pharmaceutical companies and public research organizations. As of April 2005, the NCBI parallel BLAST web server received about 400,000 BLAST queries per day [28]. These dedicated sequence search web servers often host multiple databases, with variable sizes and searched using different alignment algorithms. Meanwhile, the search workload may be highly dynamic [2].

This paper studies the scheduling strategies for parallel BLAST web servers. We reveal that a careful choice in resource allocation and database-to-processor assignment may

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 200X ACM X-XXXXX-XX-X/XX/XX ...$5.00.

easily result in an order of magnitude difference in the average query response time. The problem is that different query arrival rates, query composition, and system configurations ask for different strategies, and there are no one-size-fits-all solutions. In response, we propose a combination of *adaptive* scheduling strategies that automatically makes scheduling decisions based on the search algorithms' scalability, the storage hierarchy, and the current system load.

We evaluate our proposed techniques using a simulator verified with real cluster experiments. The results indicate that the combination of the proposed adaptive strategies achieves uniformly the best performance across a wide range of workload/system configurations, with an order-of-magnitude improvement in average query response time in many test cases.

The rest of the paper is organized as follows. Section 2 introduces background information on sequence database searches and gives the overview of our target system architecture. Section 3 presents our methodology and algorithms, while Section 4 discusses the experiment setup as well as results. Section 5 talks about related work. Finally, Section 6 concludes the paper and suggests several future work directions.

2. PARALLEL BLAST WEB SERVER ARCHITECTURE

In this section, we first briefly describe how BLAST and parallel BLAST work, then give an overview of our target parallel BLAST web server architecture.

2.1 BLAST and Parallel BLAST

The BLAST [1] family algorithms search one or multiple input query sequences against a database of known nucleotide (DNA) or amino acid sequences. The input of BLAST is one or more query sequences and the name of the target database to search. For each query sequence, BLAST performs a two-phase search using dynamic programming and returns those sequences in the database that are most similar to it. This requires a full scan of all the sequences in the database. For each of these sequences returned, BLAST reports its similarity score based on its alignment with the input query and highlight the regions with high similarity (called *hits*). Therefore, the BLAST process is essentially a top-k search, where k can be specified by the user, with a default value of 500.

Many approaches have been proposed to execute BLAST queries in parallel. Earlier work mostly adopts the *query segmentation* method [6, 8, 10], which partitions the sequence query set. This is relatively easy to implement, but cannot solve the problem caused by growing database sizes or speed up individual queries. In contrast, *database segmentation* [5, 13, 21, 22] partitions databases across processors. This approach is necessary to answer the challenge of rapidly increasing database sizes and has been gaining popularity in the BLAST user community. Hence we choose to study this parallel BLAST execution model in this paper. With database segmentation, a sequence database is partitioned into multiple *fragments* and distributed to different cluster nodes, where the BLAST search tasks are performed concurrently on different database fragments. The local results generated by individual nodes for a common query sequence are merged centrally to produce the global result.

2.2 Parallel BLAST Web Server Architecture

Figure 1 illustrates the parallel BLAST web server architecture targeted in our study, with a sample query and part of its output. As in a typical cluster setting, each node has its own memory and locally attached secondary storage, as well as access to a shared file system. One of the cluster nodes serves as the front-end node, which accepts incoming query sequences submitted online, maintains a query waiting queue, schedule the queries, and return the search results. The other nodes are back-end servers, often called "processors" in the rest of the paper for brevity.

For each query, the front-end node determines the number of processors to allocate, select a subset of idle back-end nodes (called a *partition*) when they are available, and assign these node to execute this query. After the distributed BLAST process, the search results are merged by one of the node in the partition and returned to the client via the front-end node. Note that by partitioning the cluster to run multiple queries, this architecture takes advantage of both the query- and database-segmentation models: queries are distributed to separate hardware for concurrent processing, while each query is executed in parallel through distributing the database.

To save the database processing overhead, all the sequence databases supported by the parallel BLAST web server are pre-partitioned and stored in the shared storage. Figure 1 shows two sample databases, each partitioned into 4 fragments. The required database fragments will be copied to the appropriate back-end nodes' local disk before each query is processed, and are cached there using a cache management policy. Existing parallel BLAST implementations allow multiple database fragments to be "stitched" into a larger virtual fragment with little extra overhead. Therefore for the maximum flexibility in scheduling without creating physical fragments of many different sizes, we partition the database into the largest number of fragments allowed to be searched in parallel. To simplify the scheduling and to achieve better load balance, both the database fragmentation and processor allocation are based on power-of-two numbers, which is natural considering the way clusters are purchased or built. Note that the fragments combined into a larger virtual fragment do not need to be in consecutive order. For example, when 16 processors are assigned to search a certain query against a database partitioned 64-way in a 64-processor cluster, one of them may be assigned to search fragments 0, 8, 45, and 57.

2.3 System Parameters and Assumptions

Before we move on to the scheduling strategies, we summarize important system parameters considered in our study:

- **Local storage limit**. This states how much disk storage space is available at each back-end node. The higher this limit, the more data can be cached locally for better query performance.
- **Shared storage performance**. Shared file systems equipped at clusters provide great convenience to applications, but usually have inferior performance and scalability compared to local file systems, due to contention at the interconnection networks, the file system server, or the shared disks. This is especially true for widely installed systems such as NFS [27], which were

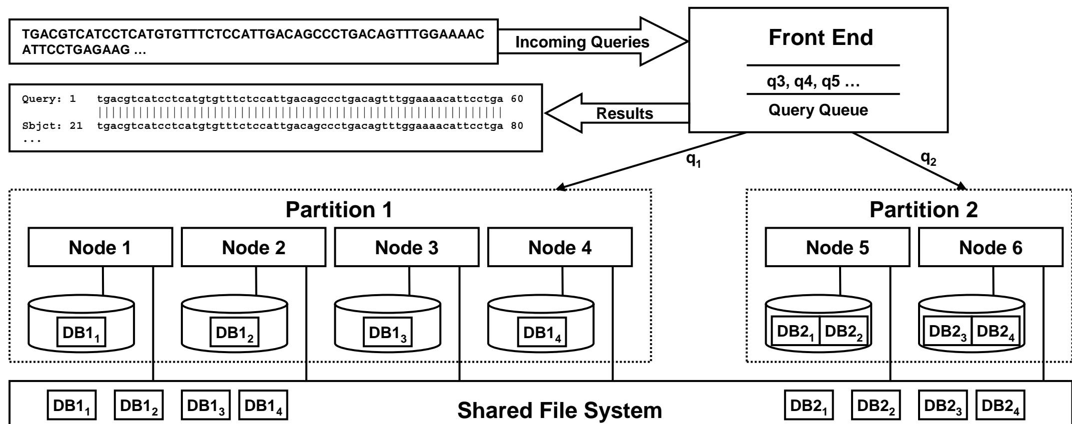


Figure 1: Target parallel BLAST web server architecture

not created for handling many concurrent large-size accesses. Particularly, we pay attention to the *relative* performance of the shared storage to the local storage, which determines the cost of local storage cache misses (when database fragments need to be copied from the shared storage).

- **Query workload**. Different query arrival rates (dense or sparse) need to be taken into account when considering the query service time vs. system throughput tradeoff. Similarly, the query composition has an impact on the data placement.

Also, we highlight our assumptions in this research. First, we assume a homogeneous environment, which is the case in most clusters. Second, to simplify our benchmarking and simulation, we assume that each query contains only one sequence to search. Although existing BLAST web servers may allow users to upload multiple query sequences, the standard NCBI BLAST engine processes input queries sequentially. The difference in search time between the shared and separate BLAST sessions for multiple query sequences is not significant and mainly lies in the initialization overhead. Our research results can be easily extended to handle multiple-sequence requests.

3. SCHEDULING STRATEGIES

In this section, we present scheduling strategies for parallel BLAST web servers. We adapt existing scheduling algorithms and propose new techniques, with an aim at designing adaptive algorithms that automatically adjust to various query workloads and cluster configurations. Our goal is to optimize the average query response time.

Scheduling in our target environment has two components. First, we perform *efficiency-oriented scheduling.* In this step, we decide how many processors to allocate for each query to be dispatched, according to the specific query workload and the current system load. Next, we perform *data-oriented scheduling.* In this step, we select a subset of processors for the query in question and determine the mapping between individual processors to database fragments, considering data locality issues. The next two sections discuss these two types of scheduling respectively.

3.1 Efficiency-Oriented Scheduling

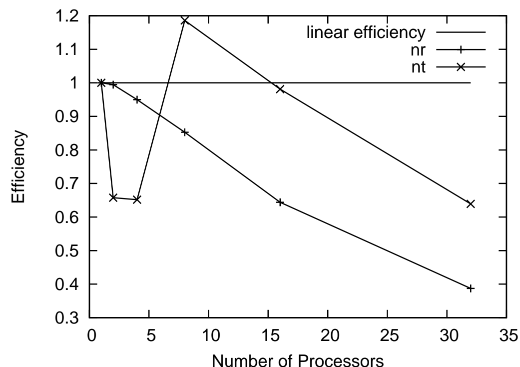


Figure 2: Parallel execution efficiency of BLAST

First, we examine parallel BLAST's performance scalability. Like most parallel applications, it is subject to the performance tradeoff between absolute performance and system efficiency when the level of concurrency is increased. One obvious explanation is the higher parallel execution overhead associated with searching a single query using more processors. Also, there is a higher degree of internal fragmentation due to load imbalance when many processors participate in a query and each of them only search a small fraction of the database. In addition, as BLAST performs top-k search, the task of processing and filtering of intermediate results grows with the number of processors. Figure 2 illustrates the performance trend of parallel BLAST from searching two widely used databases, the NCBI `nr` and `nt`, as benchmarked on our test cluster (to be described in Section 4.1). For each search workload, we plot the *efficiency*, which is defined as parallel speedup divided by the number of processors. Therefore a perfect linear efficiency is a flat line, as the result of linear speedup when the number of processors is increased. For both `nr` and `nt`, the efficiency slides steadily as more processors are used for each query.

Systems such as the NCBI BLAST server reported periodic variances in the query arrival rate [2]. One intuitive heuristic is to control the number of processors allocated to each query based on the current system load: when the

load is light, allocate more processors for smaller query response time; when the load is heavy and queries are piling up in the queue, allocate fewer processors for better system throughput (and consequently better average response time). This intuition is backed up by queuing theory and has been adopted in adaptive partitioning algorithms for parallel job scheduling [32]. In this work, we select the MAP algorithm [12], which improves upon the above work, as our base algorithm.

With MAP, both the waiting jobs and the jobs currently running are considered in determining the system load. It chooses large partitions when the load is light and small ones otherwise. More specifically, for each parallel job to be scheduled, a target partition size is calculated as

$$target_size = Max(1, \lceil \frac{n}{q + 1 + f * s} \rceil),$$

where n is the total number of processors, q is the waiting job queue length, s is the number of jobs currently running in the system, and f ($0 \leq f \leq 1$) is an adjustable parameter that controls the relative weight of q and s. In our experiments, we set the f value as 0.75, as recommended in the original MAP paper [12]. Once the target partition size is selected, the front-end node waits until these many processors become available to dispatch the query.

One may notice that in Figure 2 the `nt` curve does not monotonically decrease. Instead it peaks at 8 processors, with a super-linear speedup at that point. This is due to that the `nt` database cannot fit into the aggregate memory of 4 or fewer processors on our test platform. As BLAST makes multiple scans and random accesses to the sequence database, out-of-core processing causes disk thrashing and significantly limits the search performance. The `nr` database is much smaller and can be accommodated in a single processor's memory, therefore does not show the same behavior.

This motivates us to propose Restricted MAP (RMAP), which augments the base MAP algorithm with a database-dependent and machine-dependent memory constraint. For a given database supported by a given cluster server, we select P_{min} and P_{max}, which define the range of partition sizes (in terms of the number of processors) allowed to schedule queries against this database. P_{min} is the smallest number of processors whose aggregate memory is large enough to hold the database. P_{max} is determined by looking up the saturation point in the speedup chart: it is the largest number of processors before the absolute search performance declines. In other words, after this point deploying more processors will not produce any performance gain. An initial benchmarking is needed to set P_{max} for each database, which is feasible considering the total number of different databases supported by a web server is often moderate.

3.2 Data-Oriented Scheduling

One major motivation for parallel BLAST in the first place is that the database cannot fit into the main memory of a single computer node. As the database size increases rapidly, it is common that the combined data size is larger than the local disk space available at each cluster node. For example, in 2004 the NCBI parallel BLAST server was already serving 175GB of data, which were stored in a shared file system and copied to individual nodes on demand [2].

Like in other distributed or cluster web servers, data locality is a key performance issue in parallel BLAST web servers.

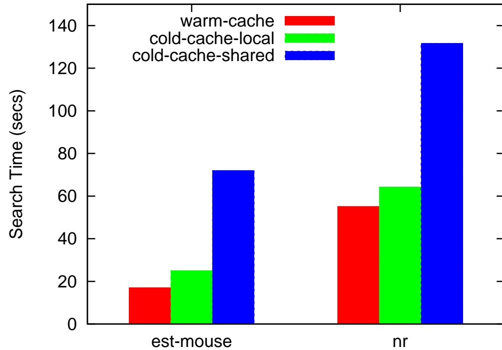


Figure 3: Impact of data placement on the BLAST performance.

Figure 3 demonstrates the impact of going down the storage hierarchy: main memory, local file system, and shared file system. The experiments use the NCBI `est-mouse` and `nr` databases, which can fit into the memory of a single processor. In the "warm-cache" tests, we warm up the file system buffer cache with the same query before taking measurements, and in the "cold-cache" tests we flush the cache. For "cold-cache-shared", we force loading the database from the shared file system. The results indicate that improving buffer caching performance and in particular, reducing remote disk accesses can significantly improve the search performance.

In this section, we discuss and propose several data scheduling strategies that enhance data locality through intelligent query scheduling and processor assignment.

These data scheduling strategies work atop the RMAP processor scheduling algorithm, which determines p, the number of processors to allocate to the query in question. A data scheduling strategy then selects the subset of p idle processors and decides the specific database fragments to assign to each processor. On each selected processor, if some of its assigned fragments are not available on the local disk, they need to be copied from the shared file system.

3.2.1 Locality-enhancing Query Scheduling

When only a part of the database fragments can be cached at each processor's local storage, and a much smaller fraction of those can be cached by the local file system in the main memory, scheduling must be performed considering the data locality issue. One intuitive locality-aware optimization is to assign queries targeting different databases to disjoint pools of processors and let each processor pool search the same database repeatedly. This way, the effective working set of each processor is reduced. Creating static per-database processor pools, however, is not flexible enough to handle the dynamic online query composition and will likely cause serious system underutilization.

A similar problem has been addressed regarding general-purpose content-serving cluster web servers. In this paper, we extend the LARD algorithm proposed by Pai et al. [29] to the parallel sequence search web server context. Given a set of back-end servers, the LARD algorithm assigns partitions of the target namespace to subsets of these servers. An incoming web request will be routed to one of the servers in its target pool, or the least loaded server if its pool is

empty. Load balancing is performed periodically to move servers from lightly loaded pools to heavily loaded ones.

Two major differences between our target system and a general-purpose cluster web server are (1) instead of a single processor, multiple processors need to be co-scheduled to queries or co-transferred between pools, and (2) a processor can handle only one query at any given time. To address this, our extended LARD algorithm establishes one global query queue (global_queue) and multiple per-database query queues (queue[DB_i]). Similarly, we have a global idle processor pool (global_pool), and multiple per-database processor pools (pool[DB_i]). Initially, all the processors are in the global pool.

Queries arrived will be appended to the global queue. A scheduling operation will be triggered by either a query arrival or a query completion. Queries in the global queue will be scheduled in the FCFS order. If the oldest query has been immediately scheduled, the algorithm will proceed to the next query in queue. Otherwise (when there are not enough resources), it blocks until the next time a query completes. This helps ensure fairness and prevents starvation. After a query is assigned to a per-database processor pool, it goes to the local queue of that pool and is scheduled using an internal scheduling algorithm (such as a fixed partitioning policy or RMAP). Algorithm 1 gives the detail of the process of scheduling one query from the global queue.

Algorithm 1 Extended LARD

```
fetch the next query q from global_queue
if pool[q.target_db] is not empty then
  append q to queue[q.target_db]
else
  partition_size ← get_recommended_size()
  m ← the number of queries waiting for q.target_db in
  global_queue
  while global_pool.size < partition_size do
    size_needed ← partition_size - global_pool.size
    find queue[DB_i] with smallest queue length
    if m > queue[DB_i].length then
      S ← release_nodes(DB_i, size_needed)
      add S to global_pool
    end if
  end while
  A ← allocate(global_pool, partition_size, q.target_db)
  add A to partition[q.target_db]
  append q to queue[q.target_db]
end if
balance_load()
```

The intuition behind the extended LARD algorithm is to assign groups of processors to processing different databases. If an incoming query does not have a processor pool assigned to its target database D, a pool will be created for D, whose size is determined by the function get_recommended_size(). This function determines how many processors to allocate, using algorithms such as RMAP. In case there are not enough processors to allocate from the global pool, the algorithm will seize processors from the most lightly loaded pool if there are fewer queries waiting in that pool's local queue than those waiting for D in the global queue. Note that the release_nodes() function may "earmark" processors that are currently running queries to be returned to the global pool once they finish. Accordingly, the allocate() function blocks until enough processors in the global queue are free.

Like in the original LARD, every time a query is scheduled the system performs load balancing. In the extended LARD, we move processors from the most lightly loaded pool (pool[DB_{min}]) to the most heavily loaded pool (pool[DB_{max}]) if one of the following conditions is satisfied:

1. queue[DB_{max}].length - queue[DB_{min}].length $>$ T and queue[DB_{max}].length $\geq 2\times$ queue[DB_{min}].length, or
2. queue[DB_{min}].length $= 0$ and queue[DB_{max}].length > 1

T in the above is a manually selected threshold, which is set as 10 in our implementation. The number of processors moved during load balancing is set to be P_min of DB_{max}, the minimum partition size allowed for the heavily loaded database. In the rest of the paper, we refer to the extended LARD algorithm as LARD.

3.2.2 Locality-aware Processor Assignment

Once we determine a target partition size to assign to a query and have a group of idle processors available, we must choose a subset of the processors to schedule the query and decide which processor is going to search which database fragments. A naive way of doing this is to order the idle processors by their processor ranks and allocate the first p of them. Similarly, the fragments are partitioned into p contiguous groups, which are assigned to the processors in the fixed order by rank. For example, suppose the database to be searched has 8 fragments, f_1 through f_8, and the query is scheduled on processors p_3 and p_{10}, then p_3 will search f_1–f_4, while p_{10} will search f_5–f_8. We call this the *first available* strategy (FA).

FA is easy to implement but it does not take into account the existing data distribution on the idle processors. Below we propose LC, an locality-aware optimization on processor assignment that tries to minimize the file copying from the shared file system. This algorithm can work together with LARD and RMAP: within the processor group assigned to each database, we use RMAP to determine the target partition size and use LC to make the fragment-to-processor assignment. While LARD reduces cache misses and data copying *implicitly* through controlling each processor's working set size, LC approaches the same goal by *explicitly* planning the data placement according to the databases' distribution on idle processors.

As mentioned in Section 2, the back-end nodes' local storage space acts as a cache space to replicate a subset of database fragments from the shared file system when the total database size exceeds the local storage limit. Since it is easy for the front-end node to keep track of which fragments are cached at each node, locality-aware scheduling algorithms can be developed to try to minimize the amount of data copied from the shared file system.

However, we have discovered that this optimization is NP-hard. Below we first give the formal problem definition and the proof regarding its complexity, then proceed to propose a greedy algorithm.

The least-copy problem: Consider scheduling a query q_i against database DB_j, which has n fragments. Suppose there are a idle processors in the system, each of which has a subset of fragments of DB_j in its local storage. Given a pre-determined partition size p, a solution to the *least-copy problem* is a mapping of the n fragments to a subset

P ($|P| = p$) of the a available processors ($p \leq a$), such that each processor in P receives n/p fragments and the total number or fragments to be copied from the shared storage is minimum. In other words, such a selection and mapping process finds a subset of processors, each of whose members contributes no more than n/p fragments so that the union of these fragments forms a maximum coverage of DB_j.

Next let us consider the decision version of the least-copy problem (DLC). Given the same input, the question is whether there exist a p-subset of the a idle processors such that an n/p-element subset from each of these p processors' local fragments exactly covers the n database fragments. Obviously this decision version has a lower complexity than that of the original least-copy problem.

We then prove the decision problem of least-copy is NP-hard, by reducing the NP-complete *Exact Cover by 3-Sets (X3C) problem* [19], defined as follows: given a set X with $|X| = 3q$ and a collection C of 3-element subsets of X, the *X3C* problem is to determine whether C contains an exact cover of X, i.e., a sub-collection $C' \subseteq C$ such that every element of X occurs in exactly one member of C'.

Proof. Given an instance of $X3C$, we construct an instance of DLC as follows. Let each fragment of DB_i correspond to an element of X and let $n = 3q$. Let $a = |C|$ and for each member C_i of C, create an available processor that has on its local storage the three fragments corresponding to C_i's content. Finally, let $p = q$. This construction can be completed within polynomial time. It is easy to see that the answer to the resulted DLC problem instance is also the answer to the original $X3C$ problem instance. □

Recognizing the complexity of the least-copy problem, we design a greedy algorithm (LC) that reduces data copy when the local storage is not large enough.

Algorithm 2 Greedy least-copy scheduling

```
U ← D
S ← ∅
n ← |D|
while |S| < p do
  for i = 1 to |I| do
    l_i = |L_{p_i} ∩ U|
  end for
  Find p_i ∈ I with the maximum l_i
  S ← S ∪ {p_i}
  F_{p_i} ← ∅
  while |F_{p_i}| < n/p and |L_{pi} ∩ U| > 0 do
    Find f_j ∈ L_{p_i} ∩ U with the minimum number of replicas among all p_k ∈ (I − S)
    F_{p_i} ← F_{p_i} ∪ {f_j}
    U ← U − {f_j}
  end while
  I ← I − {p_i}
end while
for all p_i ∈ S do
  if |F_{p_i}| < n/p then
    m ← n/p − |F_{p_i}|
    G_i ← the m-prefix of U
    F_{p_i} ← F_{p_i} ∪ G_i
    U ← U − G_i
  end if
end for
```

Algorithm 2 gives the details of the greedy LC scheduling strategy. It takes the following input parameters: $D = \{f_1, f_2, ..., f_n\}$ (the database with n fragments), $I = \{p_1, p_2, ..., p_a\}$ (a idle processors), each with $L_{p_i} \subset D$ denoting the fragments cached on its local storage, and p (the target number of processors to allocate to the current query). This algorithm iteratively selects an idle processor with the largest number of unassigned database fragments on its local storage, and assigns up to n/p cached fragments to this processor, until p processors are selected. When choosing the locally cached fragments, priority is given to the one with the fewest replicas among the unselected processors. Finally, the unassigned fragments are assigned to the selected processors in ascending order, to make sure each processor receives n/p fragments. These are "remote fragments" that must be copied from the shared file system. The output of the LC algorithm is S, the p selected processors, and F_{p_i}, the n/p distinctive database fragments assigned to each p_i in S.

3.2.3 Revisiting the RMAP Algorithm

After we consider the full storage hierarchy, during our experiments on our test cluster we have discovered a limitation of the RMAP algorithm. The problem occurs when the system load is light. When data turnover happens to occur and a large amount of data has to be copied, the slow data transfer process may significantly delay the query processing and accumulate queries in the queue. Then the system load will be considered as heavier, and the RMAP algorithm will shrink the partition size, causing additional data turnover. When the partition size stabilize at the smallest end, data turnover is reduced and the queue will quickly shorten due to the light query arrival rate. Then the RMAP algorithm begins to enlarge the partition size, resulting in repeated thrashing between the "light load" and "heavy load" modes, as well as excessive data movement.

To avoid this behavior, we propose RMAPRC, a restricted version of RMAP using a *range control* mechanism. Since the system maximum throughput of a given cluster server can be benchmarked in advance, and the actual query arrival rate can be measured at runtime using a sliding window, we let RMAPRC monitor the real system load and adjust the effective partition size range accordingly. In our implementation, RMAPRC calculates the ratio between the current query arrival rate and the system maximum throughput, and enables the corresponding fraction of partition size range starting from the upper limit. For example, suppose a certain database's partition size range is [2,16]. A load ratio of 1 will enable the full range, while a load ratio of 0.5 will enable the subrange of [8,16].

4. PERFORMANCE RESULTS

4.1 Benchmarking Configuration

In our experiments, we use five biological sequence databases downloaded from the NCBI public sequence repository. Table 1 summarizes several basic attributes of these databases. Among them, the first two are protein sequence databases (type "P") and the other three are nucleotide sequence databases (type "N"). The two types of the databases are searched using the `blastp` and `blastn` algorithms respectively. The size of each database shrinks after the database is formatted for search using the standard

Name	Type	Raw Size	Formatted Size	P_{min}	P_{max}
env_nr	P	0.25GB	0.38GB	1	8
nr	P	1.6GB	2.3GB	2	16
est_mouse	N	2.7GB	2.0GB	1	8
nt	N	17GB	5.6GB	4	16
gss	N	11GB	6.7GB	2	16

Table 1: Database characteristics

`formatdb` tool. For each of the databases, we also give the P_{min} and P_{max} pair, which defines the processor partition size range. As discussed in Section 3.1, P_{min} is determined by the memory constraint and P_{max} is determined by benchmarking the parallel execution scalability of the individual database search workload. Note that the `gss` database has a larger formatted size than `nt` does, but has a smaller P_{min}. This is because a large part of the `gss` database contains header information that does not need to be scanned during query processing.

The parallel BLAST software we used is the popular mpiBLAST tool [13, 21], available at http://mpiblast.lanl.gov/. For queries, we sampled 1000 unique sequences from the five databases, with the number of samples from each database set proportional to the database size, in terms of the number of sequences. Since sequence databases are constantly appended with newly discovered sequences, we hope this sampling method assembles the composition of real BLAST search workloads, which are driven by sequence discoveries. We compose online query traces by drawing queries randomly from this pool of unique sequences, setting the arrival interval with the Poisson distribution. Query sequences may be repeated in making longer traces.

We benchmark the query processing performance of each of the 1000 query sequences, with all the possible processor partition sizes, on a Linux cluster. The cluster has 20 compute nodes, each equipped with dual Intel Xeon 2.40GHz processors sharing 2GB of memory. Due to its target workload, this cluster has 400GB per-node local storage space, which is unusually large. The interconnection is using Gigabit Ethernet and a shared storage space of over 10TB is accessed through an NFS server.

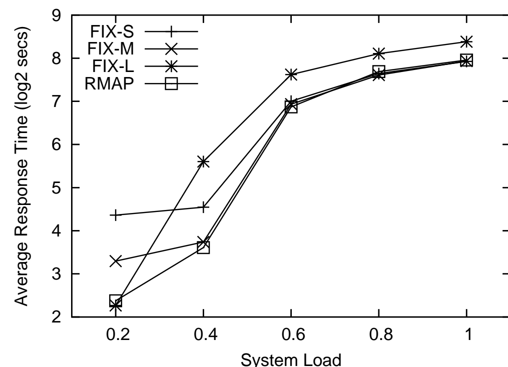
For each query, two experiments are done to measure and record a "cold-cache" and a "warm-cache" time respectively. The former has the memory flushed before a query and the latter warms up the memory cache with the same query before taking measurements.

In addition to individual query's cost, we also benchmark the maximum throughput of the whole system. This maximum throughput is calculated in an aggressive manner: we measure the maximum throughput of each database' search workload by executing the corresponding subset from the 1000-query pool on the whole cluster using the smallest partition size (P_{min}). This way the system achieves best efficiency and data locality with the single-database workload and small partition size. We then derive the multi-database maximum throughput by taking an weighted average of the single-database peak throughputs, according to the number of queries going to each database.

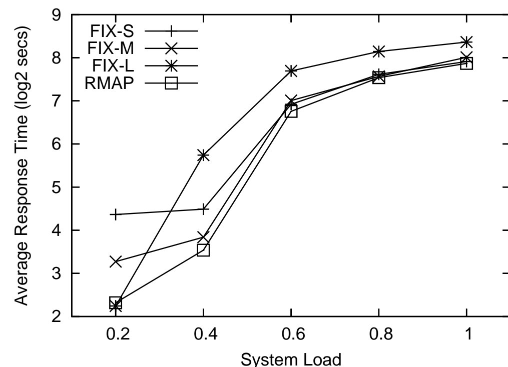
4.2 Simulation Overview

To test different cluster configurations and to accelerate experiments, we developed a parallel BLAST web server simulator. It takes as input a set of system parameters (such as the number of nodes, memory size, local disk size and shared file system performance), a query trace, as well as the processor/data scheduling algorithms, and generates the query schedule.

The cost of each individual query is calculated based on benchmarking results from the cluster described above, along with resource allocation and data distribution information. For each query sequence in our sample set, we loop up its search time with the number of processors scheduled from our benchmarking results. For partial cache hits, we approximate the cost by estimating the cache status using LRU, calculating the size of in-cache data, and performing an interpolation between the cold-cache and the warm-cache costs using that size. If certain fragments need to be copied from the shared storage, we find out the number of processors reading from the shared file system simultaneously (n) and roughly estimate the copy cost by dividing the data size by the shared file system bandwidth at n concurrent readers.



9(a) Cluster experiment results



9(b) Simulation results

Figure 4: Simulator verification results.

We verify the simulator against real experiments run on the aforementioned cluster, with the simulator configured the same way as the cluster. This test is performed with a short 300-query trace with a 200-query warm-up stage. Figure 4(a) and Figure 4(b) show the real experiment and simulation results, respectively, with several scheduling policies. Since the focus of this comparison is to verify the simulator, we leave the performance analysis to the next section

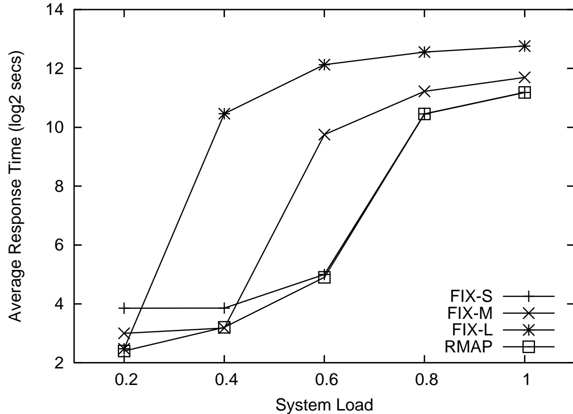


Figure 5: Results with unlimited buffer cache size.

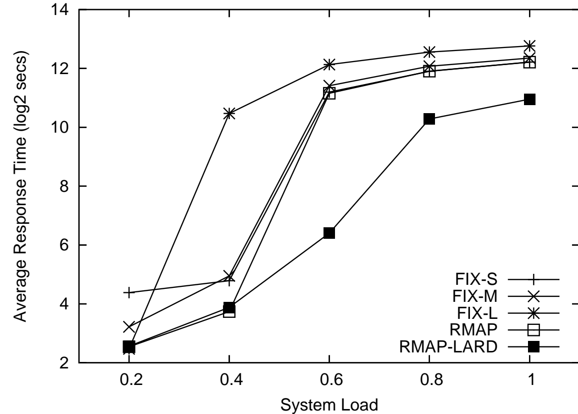


Figure 6: Results with unlimited local storage.

and instead pay attention to the accuracy of the simulation. Overall the simulation results are very close to those from real cluster runs, with a maximum error of 10.08% and an average error of 4.42%.

The rest of the paper reports results using our simulator. As in Figure 4, all the charts uses log2 scale in the y axis due to the large distribution of performance numbers. Except when noted otherwise, the simulation is performed using query traces that contain 5,000 query sequences drawn from the 1,000-sequence pool sampled from the five databases mentioned above. In addition, 800 queries are used for system warm-up.

4.3 Efficiency-Oriented Scheduling Results

To isolate the impact of parallel BLAST algorithms' scalability on the scheduling performance from that of data placement, we first simulate a situation with unlimited memory cache size. Under this scenario, after the warm-up period all the database fragments should be cached within each processor's memory. This idealization is achieved by using the search performance measured with a warm cache, where we make sure the target database fragments are cached in the main memory.

Here we compare the performance of RMAP with three versions of fixed partitioning strategies. With fixed partitioning, the number of processors allocated to queries against the same database is fixed throughout the run. For each database, we choose three fixed partition sizes within its partition size range $[P_{min}, P_{max}]$: small (FIX-S), medium (FIX-M), and large (FIX-L). Experiments are carried out using different levels of system load by adjusting the query arrival rate. A system load of 1 means the arrival rate is equal to the maximum query throughput. All the strategies here use the default FA policy in selecting idle processors to schedule and making fragment-to-processor assignments.

Figure 5 portraits the simulation results. As expected, no single fixed partitioning strategy performs consistently well. When the system load is light, the large partition size works best by using a large number of processors to reduce each query's response time. As the load increases, first the medium, then the small partition size becomes the winner. With heavier loads, smaller partition sizes help achieving better overall resource utilization by improving the parallel execution efficiency. The performance difference is significant: across the x axis, the difference between the best and worst average response time among the fixed partitioning strategies varies between 1.6 and 150 times. RMAP, on the other hand, consistently matches the best performance from the three fixed partitioning strategies by automatically adapts to the system load.

4.4 Data-Oriented Scheduling Results

We begin the evaluation of data-oriented scheduling strategies by first adding the effect of the memory buffer cache. Here we simulate the situation where the local storage space is large enough to hold all the database fragments, which is the case for certain web servers. In our experiments, by setting the local storage limit at 20GB, we allow all the database fragments to be cached at individual processors' local disk at the end of the warm-up period.

Figure 6 demonstrates the comparison between five scheduling strategies: the three fixed partitioning strategies, RMAP, and RMAP augmented with LARD. Again all of them use FA, as with sufficient local storage space it is relatively cheap to access arbitrary database fragments. While RMAP still outperforms all fixed partitioning strategies, when combined with LARD its performance is significantly improved (by up to 96% percent). By stabilizing the assignment from database to processors and reducing each processor's working set size, LARD reduces the average response time through enhanced caching. The only occasion where LARD does not help RMAP is with the system load of 0.4, where RMAP outperforms RMAP-LARD by 7%. One possible reason is that with a medium load, RMAP is likely to switch back and forth between different partition sizes, causing frequent adjustment among processor groups and a higher cache miss rate.

Note that LARD's benefit is more evident with heavier system load. This is because the RMAP algorithm will automatically shrink the partition size when the query arrival rate increases, the data footprint enlarges and smart caching becomes more important. This also explains why the FIX-L performance almost stays the same when comparing Figure 6 with Figure 5. When the partition size is large, each database is stretched thin between many processors and the data footprint is small enough to be cached in memory. Next, we investigate an extended storage hierarchy by adding shared file system accesses, and assess both

		5G-0.1	5G-0.5	5G-1.0	10G-0.1	10G-0.5	10G-1.0
Average Response Time (secs)	RMAP+FA	67925	97259	98659	8869	37795	39195
	RMAP-LARD+FA	256	314	1005	7	105	769
	RMAP-LARD+LC	71	151	786	5	59	505
Cache Misses (#misses/query/proc)	RMAP+FA	5.230	5.282	5.282	5.032	5.008	5.008
	RMAP-LARD+FA	1.842	0.342	0.042	0.128	0.787	0.056
	RMAP-LARD+LC	1.014	0.504	0.038	0.043	1.435	0.048
Copy Volume (GB)	RMAP+FA	8408	8736	8736	3317	3477	3477
	RMAP+LARD+FA	2644	289	47	124	302	31
	RMAP+LARD+LC	1179	295	37	4	242	23

Table 2: Performance impact of LARD and LC.

the LARD and LC algorithms in exploiting data locality. To simulate the case where the local storage space is insufficient to accommodate all database fragments, we impose a local storage limit to force copying data from the shared storage. Considering the total database fragment size is 17GB in our tests, we choose two storage limits at 5GB and 10GB respectively. Although these storage limits look too small for today's cluster nodes, we use them to simulate the case where the combined database size is much larger.

Table 2 shows the impact of using LARD and LC on the performance of RMAP. We performed the experiments with 3 system load levels (0.1, 0.5, and 1) and two local storage limit levels (5GB and 10GB). For each strategy, we list the average response time, the average amount of cache miss per query per processor (in terms of the number of fragments), and the total volume of data copied from the shared file system in Gigabytes.

We can see from Table 2 that LARD and LC both greatly reduce the cache misses and the shared file system accesses. In particular, by dynamically assigning different databases to disjoint sets of processors, LARD eliminates the bulk of buffer cache misses and file copying operations.

Without LARD and LC, the original RMAP algorithm is not able to keep up with the pace of query arrivals due to the small storage limit and slow shared file system, resulting in hours-long response time for each query. With LARD, however, the query response time shrinks to 5 seconds with the light system load and 10GB storage limit, and to about 13 minutes with the heavy system load and 5GB storage limit. When the local storage space is insufficient, the advantage of LARD is more evident than in Figure 6, because its data-locality-enhancing scheduling also reduces remote data copying.

LC further cuts down on the shared file system accesses, especially with the light and medium system load. This is because when the load is lighter there tend to be more idle processors at every scheduling point, allowing a larger space for the LC algorithm to play. Note that the enhanced data placement also help the buffer cache performance in many cases.

Finally, we compare our proposed combination of strategies with fixed partitioning, under the scenario of having insufficient local storage space. As we demonstrated the dramatic performance gain of using LARD and LC, the question is whether they diminish the need for the adaptive RAMP algorithm. Therefore, in the next group of experiments we use LARD and LC for all the strategies, both fixed and RAMP.

Here, we test with two shared file system bandwidth levels. The low setting, with 50MB/s peak aggregate access rate, is measured from the NFS system at our test cluster. This is a very common configuration with small- or medium-sized clusters. The high setting, with 200MB/s peak aggregate access rate, is to simulate a cluster equipped with a higher-end file system such as a parallel file system or a SAN system.

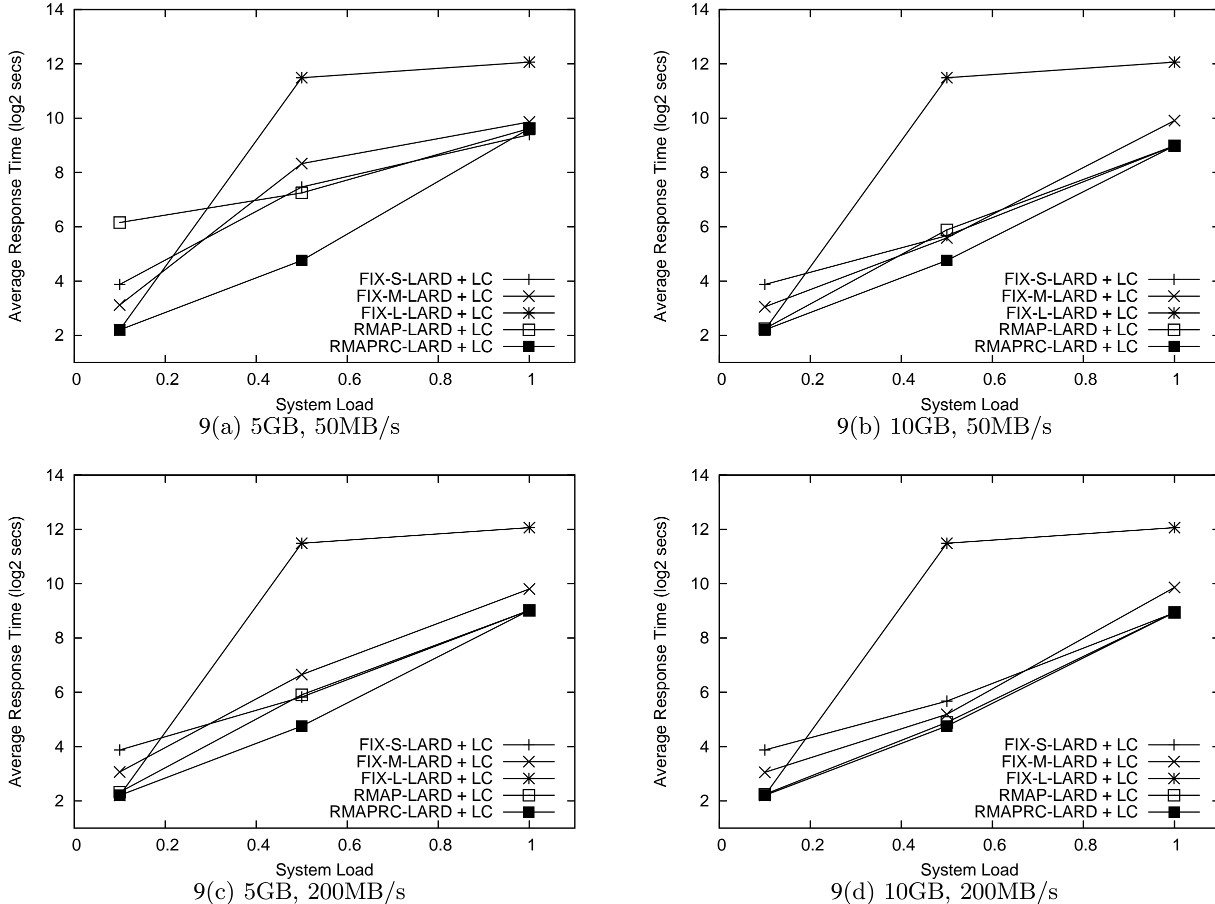
Figures 7(a) and 7(b) show the results using the low shared file system bandwidth setting, at three different system load levels (0.1, 0.5, and 1). Though not shown in the figures, the performance of each of the FIX strategies has also been greatly improved by LARD and LC. Still, no single fixed partitioning strategy always works the best. In fact, each of them may excel under certain configurations. RMAP, on the other hand, performs consistently well.

Figures 7(a) and 7(b) do illustrate the thrashing behavior discussed in Section 3.2.3: under light system load and small local storage space, the RMAP strategy may perform significantly worse than FIX-L and FIX-M. As mentioned earlier, this is due to the system oscillating between "light load" and "heavy load" when the query processing is slowed down by the database fragment copying cost. The constant adjustment of partition size, in turn, generates more data turnover. With the range control mechanism, however, this behavior is eliminated and performance is further improved from RMAP-LARD+LC. Across both storage limit levels and all system load levels, RMAPRC-LARD+LC performs the best, with up to an order-of-magnitude improvement over the best fixed partitioning strategy, and at least a 3-time improvement over the worst one.

Figures 7(c) and 7(d) repeat the above experiments, but with a much faster shared file system (200MB/s). Here the impact of data copying is much smaller compared to the previous pair of tests, but the performance benefit of our proposed optimizations is similar. Again the RMAPRC-LARD+LC performs the best, with considerable advantage over the fixed strategies.

5. RELATED WORK

Many projects have studied accelerating BLAST through parallel processing on SMP machines or clusters [5, 7, 8, 10, 13, 20, 21, 23], with the current trend of enabling database segmentation [13, 21]. Our study examines resource allocation and data placement issues related to handling online BLAST queries on a cluster web server, which can potentially work on top of any of the above underlying parallel BLAST implementations. Instead of making an individual parallel BLAST system more efficient, we focus on improving the overall resource utilization and exploiting data lo-



9(a) 5GB, 50MB/s

9(b) 10GB, 50MB/s

9(c) 5GB, 200MB/s

9(d) 10GB, 200MB/s

Figure 7: Comparison between scheduling strategies with different local storage limits and shared file system bandwidths.

cality.

There have also been studies on high throughput BLAST online services. NCBI hosts a publicly accessible BLAST server on a cluster of LINUX workstations [2, 24]. This system optimizes its caching performance by assigning a search task to the machines that have just searched the same database when possible. However, there have not been enough details released about their system design and implementation. Wang and Mu described a distributed BLAST online service system [35], where the incoming query is assigned to the least-loaded SMP computation node and each node searches one entire target database. Wang et. al. introduced a service-oriented BLAST system built on peer-topeer overlay networks [34]. This work assumes an heterogeneous environment with high communication cost. To the best of our knowledge, our paper presents the first systematic investigation, performing both processor utilization and data access optimizations, on cluster web servers providing sequence database searches.

There have been numerous studies on scalable distributed web-server systems, most of which were focused on efficient request routing and assignment for content serving, as surveyed by Cardellini et al. [9]. One closely related project is the LARD system [29], which performs content-based web request distribution to back-end servers considering both load balance and request locality for better memory cache performance. Research in this category, along with that on resource-intensive web request scheduling [36], often assumes that multiple requests can be served by the same back-end server simultaneously, or the request service time is known or can be predicted. In our target scenario, timesharing the back-end servers is difficult given the closely-coupled message passing model used by a subset of servers performing parallel BLAST, and the cost of each BLAST query has been shown as quite unpredictable [37]. Also, our work can be viewed as extending cooperative caching [11, 16, 33, 18] and coupling it with task scheduling, to support the concurrent processing of queries on multiple databases, with each query executed in parallel on multiple nodes.

Regarding space-sharing of parallel computers, a wealth of job scheduling algorithms have been proposed and evaluated, as summarized by Feitelson [17]. However, with the prevailing use of message passing programming interfaces such as MPI [25] and contemporary batch parallel job execution environments, adaptive or dynamic allocation of resources is rarely used on parallel computers. Instead, jobs are given the exact number of processors as requested, using simple strategies such as FCFS plus backfilling [26]. Our work reveals a type of real-world workload that features so called "moldable parallel jobs" (those can be run on a flexible number of processors), where many existing scheduling strategies can be applied to. In this paper, we extend existing adaptive parallel job scheduling algorithms [12, 32] to the parallel sequence database search context.

We also connected the parallel job scheduling problem with data replication and caching optimizations. To this

end, there exist studies on data replication strategies for high-performance, data-intensive applications, which mainly target the grid environment [3, 4, 30, 31]. In comparison, our target scenario is much more closely coupled and allows dynamic data partitioning that is prohibited in the grid settings due to the cost of wide-area data movement. Again, some of the above projects assumed known computation time for jobs to be scheduled, which is not true in BLAST searches.

Finally, although this paper discusses biological sequence database searches in a parallel setting, the problem is quite different from those addressed by traditional parallel databases [14, 15]. With parallel relational databases, the data distribution is far more static and handling a large number of concurrent transactions has been a standard request. In contrast, we deal with a scenario where the data can be dynamically partitioned, placed and cached, while concurrent queries are processed on multiple replicas of the databases separately.

6. CONCLUSION AND FUTURE WORK

Below we summarize the findings and contributions of this paper:

- We systematically examined the performance of online parallel bio-sequence searches, one important category of scientific database processing workloads.
- Our experiments on a real cluster revealed the large impact of both performance scalability and data locality on a parallel sequence search server.
- We extended and designed several adaptive scheduling strategies: RMAP for dynamic resource partitioning, LARD for locality-enhancing processor scheduling, and LC for optimized database-to-processor mapping. These strategies automatically react to the interplay of query workload and machine configuration, as well as the dynamic data placement along the storage hierarchy.
- We performed extensive experiments using a simulator verified with real-cluster tests. Our results demonstrate that RMAP outperforms its static counterparts across various query workloads and system configurations. Also, LARD and LC can dramatically reduce memory cache misses and data copying. Combined together, our proposed strategies often deliver an order-of-magnitude performance gain.

This work can be extended in several directions. First, we would like to study multiple-sequence queries. A related direction is to enable out-of-order execution that bundles together queries targeting the same database. Another interesting topic is to investigate intelligent data prefetching in this context.

7. REFERENCES

[1] S. Altschula, W. Gisha, W. Millerb, E. Meyersc, and D. Lipmana. Basic local alignment search tool. *Journal of Molecular Biology*, 215(3), 1990.

[2] Kevin Bealer, George Coulouris, Ilya Dondoshansky, Thomas L. Madden, Yuri Merezhuk, and Yan Raytselis. A fault-tolerant parallel scheduler for blast. In *Proceedings of the Int IEEE/ACM Super Computing Conference*, 2004.

[3] W. Bell, D. Cameron, L. Capozza, A. Millar, K Stockinger, and Floriano Zini. Simulation of dynamic grid replication strategies in optorsim. In *GRID '02: Proceedings of the Third International Workshop on Grid Computing*, London, UK, 2002. Springer-Verlag.

[4] W. Bell, D. Cameron, R. Carvajal-Schiaffino, A. Millar, K. Stockinger, and F. Zini. Evaluation of an economy-based file replication strategy for a data grid. In *CCGRID '03: Proceedings of the 3st International Symposium on Cluster Computing and the Grid*, Washington, DC, USA, 2003. IEEE Computer Society.

[5] R. Bjornson, A. Sherman, S. Weston, N. Willard, and J. Wing. TurboBLAST(r): A parallel implementation of BLAST built on the TurboHub. In *Proceedings of the International Parallel and Distributed Processing Symposium*, 2002.

[6] 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*, 17(6), 2001.

[7] R. C. Braun, K. T. Pedretti, T. L. Casavant, T. E. Scheetz, C. L. Birkett, and C. A. Roberts. Parallelization of local blast service on workstation clusters. *Future Gener. Comput. Syst.*, 17(6):745–754, 2001.

[8] N. Camp, H. Cofer, and R. Gomperts. High-throughput BLAST. http://www.sgi.com/industries/sciences/chembio/resources/papers/HTBlast/HT_Whitepaper.html.

[9] V. Cardellini, E. Casalicchio, M. Colajanni, and P. Yu. The state of the art in locally distributed web-server systems. *ACM Computing Surveys*, 34(2), 2002.

[10] E. Chi, E. Shoop, J. Carlis, E. Retzel, and J. Riedl. Efficiency of shared-memory multiprocessors for a genetic sequence similarity search algorithm. Technical Report TR97-005, University of Minnesota, Computer Science Department, 1997.

[11] M. Dahlin, R. Wang, T. Anderson, and D. Patterson. Cooperative caching: Using remote client memory to improve file system performance. In *Proceedings of the 1st USENIX Symposium on Operating Systems Design and Implementation*, 1994.

[12] S. Dandamudi and H. Yu. Performance of adaptive space sharing processor allocation policies for distributed-memory multicomputers. *J. Parallel Distrib. Comput.*, 58(1), 1999.

[13] A. Darling, L. Carey, and W. Feng. The design, implementation, and evaluation of mpiBLAST. In *Proceedings of the ClusterWorld Conference and Expo, in conjunction with the 4th International Conference on Linux Clusters: The HPC Revolution*, 2003.

[14] D. DeWitt, R. Gerber, G. Graefe, M. Heytens, K. Kumar, and M. Muralikrishna. GAMMA - a high performance dataflow database machine. In *Proceedings of the 12th International Conference on Very Large Data Bases*, 1986.

[15] D. DeWitt and J. Gray. Parallel database systems: The future of high performance database systems. *Communications of the ACM*, 35(6), 1992.

[16] M. Feeley, W. Morgan, F. Pighin, A. Karlin, H. Levy, , and C. Thekkath. Implementing global memory management in a workstation cluster. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, 1995.

[17] D. Feitelson. A survey of scheduling in multiprogrammed parallel systems. Technical Report IBM/RC 19790(87657), 1994.

[18] S. Gadde, J. Chase, and M. Rabinovich. A taste of crispy squid. In *Proceedings of the Workshop on Internet Server Performance*, June 1998.

[19] M. Garey and D. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman, San Francisco, CA, 1979.

[20] J. Grant, R. Dunbrack Jr., F. Manion, and M. Ochs. BeoBLAST: distributed BLAST and PSI-BLAST on a Beowulf cluster. *Bioinformatics*, 18(5), 2002.

[21] H. Lin, X. Ma, P. Chandramohan, A. Geist, and N. Samatova. Efficient data access for parallel BLAST. In *IPDPS '05: Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Papers*, Washington, DC, USA, 2005. IEEE Computer Society.

[22] D. Mathog. Parallel BLAST on split databases. *Bioinformatics*, 19(14), 2003.

[23] David R. Mathog. Parallel BLAST on split databases . *Bioinformatics*, 19:1865–1866, 2003.

[24] S. McGinnis and T. Madden. BLAST: at the core of a powerful and diverse set of sequence analysis tools. *Nucleic Acids Res.*, 2004.

[25] Message Passing Interface Forum. *MPI: Message-Passing Interface Standard*, June 1995.

[26] A. Mu'alem and D. Feitelson. Utilization, predictability, workloads, and user runtime estimates in scheduling the ibm sp2 with backfilling. *IEEE Transactions on Parallel and Distributed Systems*, 12, 2001.

[27] B. Nowicki. *NFS: Network File System Protocol Specification*. Network Working Group RFC1094, 1989.

[28] J. Ostell. Databases of discovery. *ACM Queue*, 3(3), 2005.

[29] V. Pai, M. Aron, G. Banga, M. Svendsen, P. Druschel, W. Zwaenepoel, and E. Nahum. Locality-aware request distribution in cluster-based network servers. In *ASPLOS-VIII: Proceedings of the 8th international conference on Architectural support for programming languages and operating systems*, 1998.

[30] K. Ranganathan and I. Foster. Identifying dynamic replication strategies for a high-performance data grid. In *GRID '01: Proceedings of the Second International Workshop on Grid Computing*, London, UK, 2001. Springer-Verlag.

[31] K. Ranganathan and I. Foster. Decoupling computation and data scheduling in distributed data-intensive applications. In *HPDC*, 2002.

[32] E. Rosti, E. Smirni, L. W. Dowdy, G. Serazzi, and B. M. Carlson. Robust partitioning policies of multiprocessor systems. *Perform. Eval.*, 19(2-3):141–165, 1994.

[33] P. Sarkar and J. Hartman. Efficient cooperative caching using hints. In *Proceeding of the 2nd USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 1996.

[34] C. Wang, B. Alqaralleh, B. Zhou, M. Till, and A. Zomaya. A BLAST service built on data indexed overlay network. *e-science*, 2005.

[35] J. Wang and Q. Mu. Soap-HT-BLAST: high throughput BLAST based on Web services. *BIOINFORMATICS -OXFORD-*, 2003.

[36] H. Zhu, B. Smith, and T. Yang. Scheduling optimization for resource-intensive web requests on server clusters. In *Proceedings of the 11th ACM Symposium on Parallel Algorithms and Architectures*, 1999.

[37] Details omitted due to double-blind reviewing.