

Adaptive Request Scheduling for Parallel Scientific Web Services

Heshan Lin¹, Xiaosong Ma^{1,2}, Jiangtian Li¹, Ting Yu¹, and Nagiza Samatova^{1,2}

¹ Department of Computer Science, North Carolina State University

² Computer Science and Mathematic Division, Oak Ridge National Laboratory

Abstract. Scientific web services often possess data models and query workloads quite different from commercial ones and are much less studied. Individual queries have to be processed in parallel by multiple server nodes, due to the computation- and data-intensiveness of the processing. Meanwhile, each query is performed against portions of a large, common dataset. Existing scheduling policies from traditional environments (namely cluster web servers and supercomputers) consider only the data or the computation aspect alone and are therefore inadequate for this new type of workload.

In this paper, we systematically investigate adaptive scheduling for scientific web services, by taking into account parallel computation scalability, data locality, and load balancing. Our case study focuses on high-throughput query processing on biological sequence databases, a fundamental task performed daily by millions of scientists, who increasingly prefer to use web services powered by parallel servers. Our research indicates that intelligent resource allocation and scheduling are crucial in improving the overall performance of a parallel sequence database search server. Failure to consider either the parallel computation scalability or the data locality issues can significantly hurt the system throughput and query response time. Also, no single static strategy works best for all request workloads or all resources settings. In response, we present several dynamic scheduling techniques that automatically adapt to the request workload and system configuration in making scheduling decisions. Experiments on a cluster using 32 processors show the combination of these techniques delivers a several-fold improvement in average query response time across various workloads.

1 Introduction

There is a growing trend to provide parallel scientific computation services through the web interface, especially for computation- and data-intensive tasks such as scientific database queries, data mining, and visualization. Rather than having users download large volumes of shared data and run stand-alone applications, scientific web services allow them to perform common data processing/analysis tasks through intuitive web interfaces. For example, an online bio-sequence search service can be viewed as the equivalent of web search engine in the bioinformatics world.

These services are very appealing to scientists due to several reasons. First, for many researchers the existence of web data processing services reduces or even eliminates the purchase and maintenance cost of owning local clusters. Second, many popular data processing tasks access shared public datasets (such as well-known sequence databases or satellite images) that are constantly updated. Having such datasets managed by a parallel web server enables individual scientists to access the latest data without worrying about downloading, storing, and updating large datasets. Last but not least, providing parallel scientific data processing through transparent and intuitive web interfaces hides the painful details of parallel computing. It lets domain scientists obtain the performance of powerful clusters without dealing with tedious and challenging tasks such as machine administration, batch job submission and monitoring, manual data staging, and after all, learning or even writing parallel software.

Given a computing platform (typically a cluster of back-end servers and one or more frontend servers), a collection of shared datasets, and a collection of applications to run on demand as services, efficient scheduling is crucial to the parallel web server's performance, in terms of the average request response time. However, existing scheduling strategies from two related application fields, namely commercial clustered web servers and space-shared parallel computers, are inadequate for this new type of workload. Below we briefly describe the reasons (more detailed discussion will be given in Section 5).

Scientific web services tend to be both computation- and data-intensive, performing non-trivial algorithms over large amounts of shared data. In contrast, commercial web servers typically stream contents or perform low-cost relational database queries. Hence their scheduling algorithms concentrate on data locality optimization and load balancing. Also, a back-end server node usually handles many client requests simultaneously with multiple open connections. With scientific data analyzing services, the CPU and the memory resources required to timely process a request are often far beyond those can be offered by a single node. Consequently, a group of these nodes is dedicated to every request in a tightly synchronized manner.

In this sense, request processing in scientific web services is closer to batch job processing on parallel computers, however with two major differences. First, on general-purpose parallel computers, batch jobs are mutually independent and rarely share data. Second, as shared computation platforms, parallel computers have no knowledge regarding each job's computation and I/O requirements, and the resources requested by each job (such as the number of processors and the maximum run time) are specified explicitly in job scripts. Therefore batch job scheduling usually pays no attention to data locality issues and has no control over the level of concurrency in each job. With scientific data services hosted by specialized data centers, data sharing is common and the parallel web server has much more knowledge about the services it provides.

Therefore, parallel scientific web services require careful examination of the intertwined computation and data management issues in making scheduling decisions. In this paper, we extended scheduling algorithms to work for parallel

scientific web services, from those designed for the commercial cluster web servers and batch processing parallel computers. By adopting a novel combination of these extended algorithms, a parallel scientific web server will take into consideration both the data and the computation aspects: data locality, parallel execution efficiency, and load balancing. In addition, the combined strategies work fully adaptively, automatically adjusting scheduling strategies according to the server load levels and dynamic data access patterns.

We implemented our proposed scheduling algorithms, along with baseline strategies to compare with, in a parallel BLAST server prototype. BLAST [1] is a fundamental sequence database search task performed routinely by scientists. Given a query sequence, the BLAST family tools search through a database of known sequences and return sequences that are “similar” to it. Online parallel BLAST searches have become popular. In April 2005, the NCBI parallel BLAST web server received about 400,000 BLAST queries daily [19]. Such dedicated sequence search web servers often host multiple databases and provide different alignment algorithms. Meanwhile, the search workload is highly dynamic [2].

Our experiments on a cluster server performing parallel BLAST revealed that a careful choice in query concurrency and database-to-processor assignment may easily result in a dramatic difference in the average query response time. We confirmed that different query arrival rates and query composition ask for specialized strategies, and there are no “one-size-fits-all” solutions. The combination of the proposed adaptive strategies, however, achieves the best or close-to-best performance across a wide range of system load levels, with a several-fold improvement in average query response time in many cases.

Although our implementation and evaluation are based on parallel BLAST, this workload carries many common characteristics of scientific data processing applications, such as accessing large shared databases, map-reduce type of processing, and content-dependent execution time. We believe that the observations and experiences collected through this study can be utilized by many other applications.

2 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 heuristics based, two-phase search on all sequences in the database and returns those 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 highlights 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. Among them, the *database segmentation* [3,9,13,15] model has proved to be

effective in processing the ever growing sequence data sets. 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 results.

2.2 Parallel BLAST Web Server Architecture

Figure 1 illustrates the parallel BLAST web server architecture targeted in our study, with sample query and partial output. As in a typical cluster setting, each node has its own memory and local disk 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, schedules the queries, and returns 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, selects a subset of idle back-end nodes (called a *partition*) when they are available, and assigns these nodes to execute this query. After the parallel BLAST search, the results are merged by one of the nodes in the partition and returned to the client via the front-end node.

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

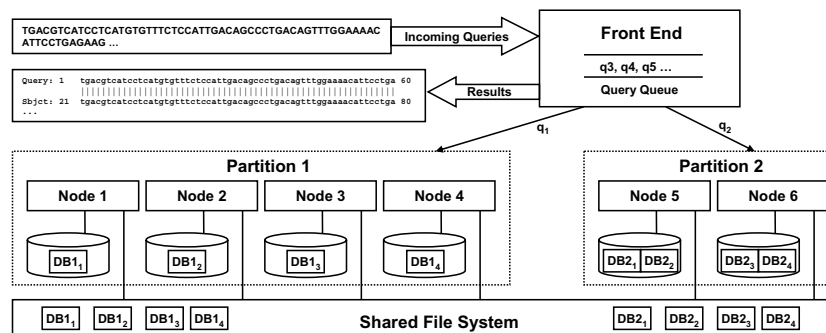


Fig. 1. Target parallel BLAST web server architecture

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.

Assumptions: Before we move on to the scheduling strategies, we summarize assumptions made in this study: First, we assume a homogeneous environment, which is true for most clusters. Second, due to the space constraint, in this paper we discuss the scenario where the entire collection of databases can be accommodated at each cluster node’s local disks.¹ This is likely the case for parallel BLAST servers, as the total size of formatted NCBI sequence databases is currently around 100GBs, while a cluster node can easily have hundreds of GBs of local disk space today. Finally, to simplify query workload generation, we assume that each query contains only one sequence. 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. In the rest of the paper, we use the terms “request” and “query” interchangeably.

3 Scheduling Strategies

In this section, we present scheduling strategies for parallel scientific web services, using parallel BLAST server as a case study. We extend two existing scheduling algorithms and integrate them to design adaptive algorithms that automatically adjust to various query workloads and cluster configurations. Like in many existing request scheduling studies, our major goal is to optimize the average query response time.

In Section 3.1 and Section 3.2, we discuss our extended scheduling algorithms respectively. The first one comes from the commercial cluster web server community and performs *data-oriented scheduling*. It determines which processors should be allocated for a specific query, considering existing data cached at these processors and their current load. The second one comes from the space-sharing parallel job scheduling community and performs *efficiency-oriented scheduling*. It determines the desired level of concurrency for processing a query, considering the specific query workload and the current system load. Both algorithms are extended substantially to fit the scenario of parallel scientific web services. Then in Section 3.3, we discuss our overall scheduling scheme and describe how we integrate the two scheduling algorithms.

¹ For systems equipped with insufficient local storage, we have developed additional optimizations, as described in our technical report [14].

3.1 Data-Oriented Scheduling

Like in other distributed or cluster web servers, data locality is a key performance issue in parallel BLAST web servers. Figure 2 demonstrates the impact of going down the storage hierarchy: main memory, local file system, and shared file system. The experiments use sequential NCBI BLAST to search the `est-mouse` and `nr` databases, which can fit into the memory of a single processor. For each case, 10 sequences randomly sampled from the database itself are used as queries, and the average search time is reported. 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 first. For “cold-cache-shared”, we force loading the database from the shared file system. The results indicate that improving file caching performance and in particular, reducing remote disk accesses can significantly improve the search performance.

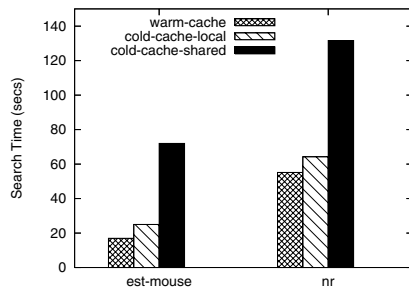


Fig. 2. Impact of data placement on the BLAST performance

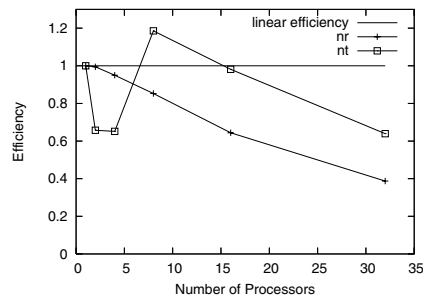


Fig. 3. Parallel execution efficiency of BLAST

As mentioned earlier, in this paper we focus on the scenario where the entire set of databases hosted by a parallel scientific web server can fit into the per-node local disk space. Still, only a small fraction of those databases can be buffer-cached in the main memory, and 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 for content request distribution proposed by Pai et al. [20] to the parallel scientific web service context. Given a set of back-end servers, the LARD algorithm assigns partitions of hosted targets to subsets of these servers. An incoming web request will be routed to one of the servers assigned to its target, or the least loaded server if it is the first request of the given target. Load balancing is performed

periodically to move requests from heavily loaded servers to lightly loaded ones. LARD exploits data locality to improve the server performance by assigning requests of the same target to the same set of processors.

Two major differences make our target system considerably more complex than a general-purpose cluster web server. First, multiple processors need to be co-scheduled to queries or co-transferred between pools. Second, a processor can handle only one query at any given time. Therefore queries cannot be piled to server nodes as they arrive, but need to wait for dispatch.

To handle these requirements, we extend LARD to a new algorithm called PLARD (Parallel LARD). To perform locality-aware assignment and load balancing, PLARD adopts a two-level scheduling mechanism. It establishes one global query queue (`global_queue`) and multiple per-database query queues (`queue[DBi]`). Queries will be first appended to the global queue, and subsequently dispatched to one of the per-database queues. Similarly, because servers need to be assigned in groups, PLARD manages a global idle processor pool (`global_pool`), and multiple per-database processor pools (`pool[DBi]`). Initially, all the processors are in the global pool. A scheduling operation will be triggered by either a query arrival or a query completion. Algorithm 1 gives the detail of the process of scheduling one query from the global queue.

Queries in the global queue will be scheduled in the first-come-first-serve (FCFS) order. When there are not enough resources for the next query, the scheduling attempt is aborted and the global scheduler waits until a query completes. This helps ensure fairness and prevents starvation. Also, this allows the recommended partition size to be recalculated as the system load changes.

Before moving a query from the global queue to a per-database pool, a recommended partition size will be calculated by the function `get_recommended_size()`. This function determines how many processors should be allocated to a target database, using algorithms such as the ones described in the next section. The target database-pool will be enlarged if the pool size is less than the recommended size. 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 the target database in the global queue.

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, as presented in the next section). This way, a relatively stable subset of server nodes are assigned to work on a certain database, maximizing the use of their collective buffer cache space.

Like in the original LARD, every time a query is scheduled the system performs load balancing. In PLARD, we move processors from the most lightly loaded pool (`pool[DBmin]`) to the most heavily loaded pool (`pool[DBmax]`), if one of the following conditions is satisfied:

1. `queue[DBmax].length - queue[DBmin].length > T` and `queue[DBmax].length ≥ 2 × queue[DBmin].length`, or
2. `queue[DBmin].length = 0` and `queue[DBmax].length > 1`

Algorithm 1. PLARD

```

fetch the next query  $q$  from global_queue
 $partition\_size \leftarrow get\_recommended\_size()$ 
 $m \leftarrow$  the number of queries waiting for  $q.target\_db$  in global_queue
 $candidate\_queues \leftarrow \bigcup queue[DB_i]$ , where  $DB_i$  not equal to  $q.target\_db$ 
 $increase\_size \leftarrow partition\_size - pool[q.target\_db].size$ 
if  $increase\_size > 0$  then
  while global_pool.size <  $increase\_size$  and candidate_queues not empty do
     $size\_needed \leftarrow increase\_size - global\_pool.size$ 
    find queue[DBj] ∈ candidate_queues with smallest queue length
    if  $m > queue[DB_j].length$  then
       $num\_idle \leftarrow$  the number of idle nodes in pool[DBj]
       $S \leftarrow release\_idle\_nodes(DB_j, min(num\_idle, size\_needed))$ 
      add  $S$  to global_pool
    end if
    remove queue[DBj] from candidate_queues
  end while
  if  $increase\_size \leq global\_pool.size$  then
     $A \leftarrow$  allocate  $increase\_size$  processors from global_pool
    add  $A$  to pool[q.target_db]
  end if
end if
if pool[q.target_db] is not empty then
  append  $q$  to queue[q.target_db]
end if
balance_load()

```

T in the above is a configurable 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} , where P_{min} is the minimum partition size allowed for a given database as described in Section 3.2. This helps reduce the internal fragmentation of a database pool during load balancing.

3.2 Efficiency-Oriented Scheduling

PLARD helps us optimize query processing performance by maximizing the use of cached data and improving load balance between server nodes. However, it does not consider the parallel processing scalability of the scientific applications that service the web requests. The latter turns out to be crucial in deciding how many processors should be allocated to each individual query, and can have a significant impact on the parallel web server's performance.

We illustrate the argument by examining 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 increases. One obvious explanation is the higher parallel execution overhead associated with searching a single query using more processors. In addition,

as BLAST performs top-k search, the task of processing and filtering of intermediate results grows with the number of processors. Figure 3 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. 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 [21]. In this work, we select the MAP algorithm [8], 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 [8]. 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 3 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 compute node'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 moderate².

For each query scheduled, when there are more idle processors available than p , the desired partition size calculated, RMAP adopts a simple node selection strategy called FA (First Available), where the first p idle processors by the processor rank will be assigned to work on the query. Database fragments will be assigned to these processors in a round-robin manner.

3.3 Combining PLARD and RMAP

We integrated PLARD and RMAP in our two-level query scheduler implementation for the parallel BLAST server prototype.

As shown in Algorithm 1, when dispatching a query from the global queue to a particular DB queue, the RMAP algorithm is first used to calculate a recommended partition size based on the global system state. More specifically, the queue length(q) is calculated by summing up all queries in the global queue and local DB queues, and the number of queries in the system(s) is the sum of queries being searched at all DB pools. If the number of idle processors in the processor pool of the target DB is smaller than the recommend partition size, the scheduling algorithm seeks to assign more processors to this pool by acquiring idle processors from the system idle processor pool and/or other relatively lightly-loaded DB pools.

When a partition with the recommended size can be provided, the query is moved into the local DB queue. There the RMAP algorithm will be called again to determine a proper partition size in local scheduling. At this point, each local RMAP scheduler uses the local system state, namely the local DB queue length as q and the number of queries being serviced in the local processor pool as s .

With this two-level scheduling approach, we adapt simultaneously to the intensiveness and the database access pattern of the dynamic query workload by leveraging strengths of both RMAP and PLARD. The two algorithms complement each other nicely under the new scheduling framework.

4 Performance Results

4.1 Experiment 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`

² The number of all sequence databases offered by the NCBI web search is 21 at the time this paper is written.

Table 1. Database characteristics. Note the P_{min} values are multiples of 2, this is because our experiments are performed on a two-way SMP cluster, and we found using a compute node (2 processors) as the smallest scheduling unit yields better performance than does using an individual processor, as the former choice has better data locality.

Name	Type	Raw Size	Formatted Size	P_{min}	P_{max}
env_nr	P	1.7GB	2.5GB	2	32
nr	P	2.6GB	3.0GB	4	32
est_mouse	N	2.8GB	2.0GB	2	16
nt	N	21GB	6.5GB	8	32
gss	N	16GB	9.1GB	8	32

algorithms respectively. The size of each database shrinks after the database is formatted for search using the standard `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 3.2, 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.

The parallel BLAST software we used is the popular mpiBLAST tool [9,13], available at <http://mpiblast.org/>. For queries, we sampled 1000 unique sequences from the five databases, with the number of samples from each database proportional to the formatted database size. Since sequence databases are constantly appended with newly discovered sequences, we hope this sampling method resembles 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.

To create traces with the desired arrival rates, we 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's 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 a weighted average of the single-database peak throughput, according to the number of queries going to each database.

Unless noted otherwise, the experiments are performed using query traces that contain 600 query sequences sampled from the 1000-query pool above. Note that many of the charts use log2 scale on the y axis, due to the large distribution of performance numbers under different system load levels.

4.2 Test Platform

Our experiments were performed on the Orbitty Linux cluster located at North Carolina State University. Orbitty consists of 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 large enough to host the entire collection of NCBI sequence databases. The interconnection is Gigabit Ethernet and a shared storage space of over 10TB is accessed through a Lustre server.

4.3 Data-Oriented Scheduling Results

First, we examine the effectiveness of improving data locality in query processing, by showing the impact of PLARD on 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 query arrival rate is equal to the maximum query throughput. All these strategies also use the default FA policy in selecting idle processors to schedule.

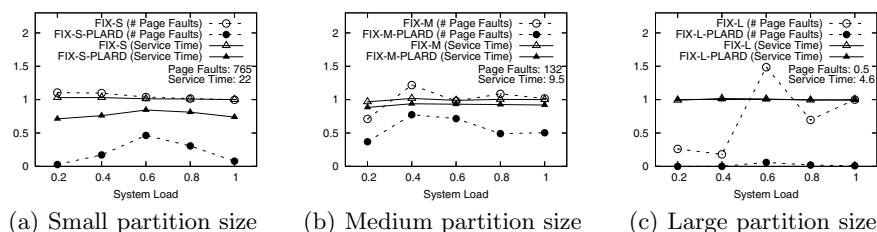


Fig. 4. Normalized average number of page faults and normalized average service time

Figure 4 portrays the impact of PLARD on the fixed partition size algorithms' file caching performance. Since BLAST uses memory mapped files, the number of page faults is a good indication of the amount of file I/O performed to retrieve the database fragments. For each of the fixed algorithms, we plot the average number of page faults per node (dashed lines) and the average query service time (solid lines), with and without PLARD. All the page fault numbers are normalized against the page fault number of the original algorithm (without PLARD) with the system load of 1. The same applies to the service times. The absolute values of these two pivot numbers are marked in the charts.

As expected, the PLARD algorithm does have a significant impact on the number of page faults. In particular, for FIX-S, the original page fault numbers of over 750 are reduced at least by half, and almost eliminated with the lightest and heaviest system loads. On average, the number of page faults is reduced by 79.87%. The original FIX-S page fault slightly declines as the system load intensifies since more processors will be actively used, and the chance of having cache hits increases due to the enlarged aggregate memory size, although there is no intentional, locality-aware query placement. With PLARD, however, the peak of page fault numbers appear in the medium load (0.6), where with the

small partition size, the per-database processor pools are the most dynamic: processors are shifted between pools relatively more frequently, reducing the chances of cache hits within each database pool.

With FIX-M, the page fault reducing of PLARD is smaller but still considerable, with an average of 43.37% decrease. Here the peaks of the page fault numbers, both with and without PLARD, are different from those with FIX-S due to the larger partition sizes. For example, the lightest load achieves the best data locality since the query load is rather concentrated on a group of processors, facilitating in-memory data reuse, while the size of the group is large enough to spread the databases out and reduce the data access working set per node.

With FIX-L, the databases are so spread out so that all the fragments needed by a processor are almost always in the memory. Although the normalized curves look dramatic, the absolute numbers are very small. Even without PLARD, the cache misses are negligible, with an average page fault count of 0.37.

The improvement of service times using PLARD is a direct result of the improved data locality, as PLARD does not affect the computation efficiency of each query's processing with the fixed-partitioning algorithms. The degree of the improvement, however, declines as the partition size selected increases. This is because the number of page faults goes down faster than the service time does when larger partitions are used. Therefore the impact of page fault reduction plays a smaller and smaller role.

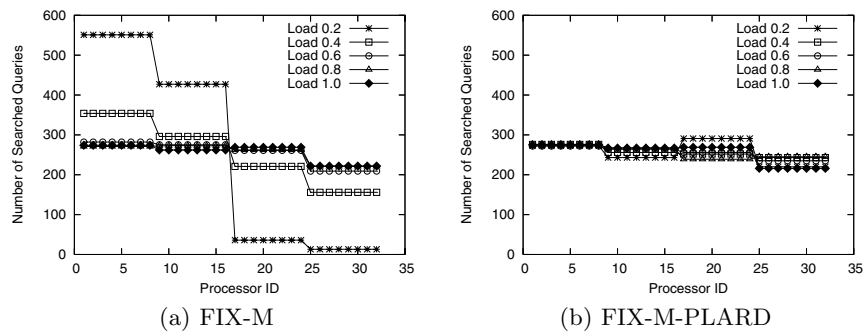


Fig. 5. Query load distribution among processors with the medium partition size

Figure 5 provides additional information about the effect of PLARD, from its load balancing aspect. We illustrate this using FIX-M, the algorithm using the medium partition size. As discussed above, with the FA policy for processor assignment, the query processing workload distribution is skewed at the load level of 0.2. Most queries are assigned to the first 16 processors, with an additional 100+ queries assigned to the first 8 (please recall that the “medium partition size” varies from database to database). Heavier system loads force the queries to become more evenly distributed. With PLARD, the query processing assignments are well balanced among processors for all system load levels.

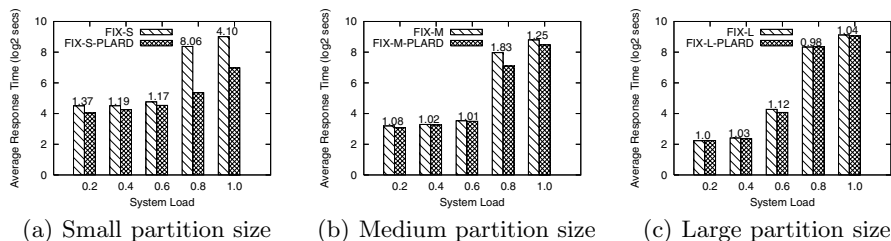


Fig. 6. Impact of PLARD on the average query response time. Note that the y axis uses the log2 scale, and the speedup factor brought by PLARD is shown at the top of each pair of bars.

Now we take a look at the overall impact of PLARD, by comparing the average query response time before and after. Because long waiting time with heavy system loads caused a wide distribution of response time, we show the numbers in log scale, with the speedup factor brought by PLARD labeled at the top of each pair of bars.

Figure 6 shows the comparison, again for each of the FIX algorithms using multiple system loads. As expected, the largest improvements are found with FIX-S, where the average response time is reduced by up to 4.1 times. As we have seen from Figure 4, the largest enhancement to data locality and the average query service time occurs with the small partition size. The changes in service time, in turn, has a varying impact on the query response time. With heavier loads, the reduced service time has a rather dramatic effect on decreasing the queue length and average query wait time. With light loads, the enhanced service time does not affect the per-query wait time much. For FIX-M, the best improvement is observed at the load of 0.8, with a speedup factor of 1.83. Not surprisingly, PLARD does not bring significant improvement to FIX-L.

4.4 Efficiency-Oriented Scheduling Results

Now we examine the impact of RMAP by enabling PLARD for all tests and compare the three FIX algorithms with RMAP.

Figure 7 portrays the 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 3.5 and 8 times. RMAP, on the other hand, closely matches the best performance from the three fixed partitioning strategies by automatically adapting to the system load.

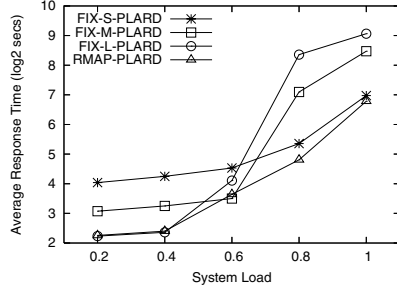


Fig. 7. Performance of combined RMAP and PLARD with fixed arrival rates (y axis uses log₂ scale)

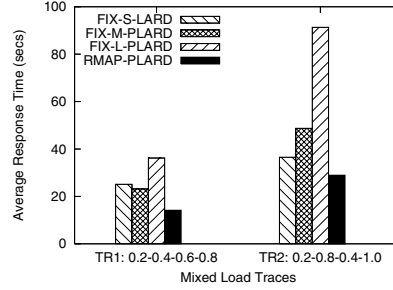


Fig. 8. Performance of combining RMAP and PLARD on two 800-sequence traces with mixed arrival rates

The only point where the RMAP performance is visibly lower than the best fixed partition size algorithm is with the medium system load (0.6). Because the trace we used is not even-paced, the medium load is an unstable case for RMAP, where the scheduler adjusts the partition size (in both directions) most frequently. With frequent partition size changes, cache contents cannot be well utilized and more cold misses are introduced.

To verify this, we take a closer look at the behavior of FIX-M-PLARD and RMAP-PLARD. Table 2 summarizes a group of measurements taken from the experiments using the two algorithms, at the system load level 0.6 and 0.8. Because the partition sizes are power-of-two numbers, we calculate the average partition size by taking the arithmetic average after performing the log₂ operation. The “total service time” is calculated as the total computation resource usage in an experiment. For each query, we calculate its resource usage as the product of its service time and the number of processors it used. We sum up the resource usage of all queries in a trace as the total service time.

From the page fault counts, we see that RMAP does hurt the data locality at load level 0.6. Consequently, RMAP adopts a slightly larger partition size than FIX-M does, but has a 8% higher average service time. The service time increase causes a similar increase in the waiting time and average response time.

Interestingly, RMAP caused a much larger increase in the number of page faults at the load of 0.8, yet the average response time of RMAP is 5 times better than that of FIX-M. This is caused by that RMAP has better parallel

Table 2. FIX-M-PLARD and RMAP-PLARD statistics at system load 0.6 and 0.8

System Load	0.6		0.8	
Policy	FIX-M-PLARD	RMAP-PLARD	FIX-M-PLARD	RMAP-PLARD
Average # Page Faults	93.00	132.74	63.64	224.88
Average Service Time (s)	8.81	9.53	8.76	15.47
Average Waiting Time (s)	2.50	2.95	128.25	12.51
Average Response Time (s)	11.31	12.48	137.02	27.98
Total Service Time (s)	69936	78700	69657	63977
Average Partition Size (log ₂)	3.71	3.88	3.71	2.78

computation efficiency there, which can be seen from the total service time: RMAP increased the total service time at 0.6 and decreased it at 0.8. Although the individual query's service time is longer than FIX-M, RMAP increases the whole-system throughput by automatically adopting a considerably smaller average partition size. With such a heavy system load, this had a dramatic effect on shortening the average query waiting time, and the average query response time consequently.

Finally, we evaluate the overall adaptivity of the combined RMAP-PLARD algorithm. Figure 8 shows two sets of experiments, each using a mixed load level trace containing 800 queries. In each trace, the average load level is adjusted several times, *e.g.*, from 0.2 to 0.4, 0.6, and finally 0.8, for four equal-length intervals (in terms of the number of queries). Trace 1 adopts such a monotonically rising system load as in the above example, while trace 2 has a repeated up-down pattern. Again, for such mixed load traces, none of the fixed partition size algorithms consistently win, and each of them may suffer trace intervals where the selected partition size is undesirable. RMAP, on the other hand, successfully adapts to the varying query intensiveness and significantly outperforms all the fixed partition size algorithms, bringing an improvement factor of 1.63 and 1.26 in average response time over the best performing fixed algorithm for trace 1 and 2, respectively.

5 Related Work

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. [6]. One closely related project is the LARD system [20], 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 [24], 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, time-sharing the back-end servers is difficult given the closely-coupled message passing model used by a subset of servers performing parallel scientific applications, and the cost of each request could be quite unpredictable [11].

Regarding space-sharing of parallel computers, a wealth of job scheduling algorithms have been proposed and evaluated, as summarized by Feitelson [10]. However, with the prevailing use of message passing programming interfaces such as MPI [17] 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 strategies such as FCFS plus backfilling [18]. 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 [8,21] to the high-performance scientific web service context.

Many projects have studied accelerating BLAST through parallel processing on SMP machines or clusters [3,4,5,7,9,12,13,15], with the current trend of enabling database segmentation [9,13]. Our case study of parallel BLAST server examines resource allocation and data placement issues related to handling on-line 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 locality.

There have also been studies on high throughput BLAST online services. Wang and Mu described a distributed BLAST online service system [23], where the incoming query is assigned to the least-loaded SMP node and each node searches one entire target database. Wang et. al. introduced a service-oriented BLAST system built on peer-to-peer overlay networks [22]. This work assumes a heterogeneous environment with high communication cost. NCBI hosts a publicly accessible BLAST server on a farm of LINUX workstations [2,16]. For a given query, the system statically splits the search into 10-20 subtasks, each searching a different piece of the database. The subtasks are scheduled independently to the machines that have just searched the same piece of data when possible. A central machine tracks and merges results from subtasks for all queries. Due to the lack of design/implementation details about the NCBI BLAST server in the literature, we were not able to do a direct comparison. However, we argue that the NCBI server is not able to factor in the parallel efficiency by using only static task partitioning. To the best of our knowledge, our paper presents the first systematic investigation of optimizing scientific web services by taking into account both parallel efficiency and data locality.

6 Conclusion and Future Work

Below we summarize the findings and contributions of this paper: **(1)** We identified the scheduling requirements of increasingly popular parallel scientific web services. **(2)** For our target workload, we extended and designed several adaptive scheduling strategies, namely PLARD for locality-enhancing resource partitioning, and RMAP for dynamic parallelism adjustments. These strategies automatically react to the query workload, both in terms of the request intensiveness and the data access pattern. **(3)** We integrated and implemented our proposed algorithms in a parallel BLAST sequence search prototype and performed extensive experiments using real-cluster tests. **(4)** Our results demonstrated that PLARD can significantly reduce the amount of file I/O. Meanwhile, RMAP outperforms its static counterparts across various query workloads. Combined together, our proposed strategies often deliver an several-fold performance gain.

This work can be extended in several directions. First, we would like to apply our scheduling algorithms to other parallel web services such as online scientific data mining. Another interesting topic will be investigating how to combine

intelligent data prefetching and our adaptive scheduling to host service of huge datasets that cannot be fully cached in a server node's local storage.

Acknowledgment

This research has been supported by a DOE ECPI Award (DE-FG02-05ER25685), an NSF CAREER Award (CNS-0546301), and NSF grants CCF-0621470 and IIS-0430166. In addition, the work is supported by the joint appointment between NCSU and ORNL for Xiaosong Ma and Nagiza Samatova, and by the Scientific Data Management Center (<http://sdmcenter.lbl.gov>) under the Department of Energy's SCIDAC program. We thank Dr. John Blondin at North Carolina State University for facilitating our experiments on the Orbitty cluster, and we thank Andrew Brown for providing technical supports on the cluster.

References

1. Altschula, S., Gisha, W., Millerb, W., Meyersc, E., Lipmana, D.: Basic local alignment search tool. *Journal of Molecular Biology* 215(3) (1990)
2. Bealer, K., Coulouris, G., Dondoshansky, I., Madden, T., Merezhuik, Y., Raytselis, Y.: A fault-tolerant parallel scheduler for blast. In: SC 2004 (2004)
3. Bjornson, R., Sherman, A., Weston, S., Willard, N., Wing, J.: TurboBLAST(r): A parallel implementation of BLAST built on the TurboHub. In: IPDPS (2002)
4. Braun, R.C., Pedretti, K.T., Casavant, T.L., Scheetz, T.E., Birkett, C.L., Roberts, C.A.: Parallelization of local blast service on workstation clusters. *Future Gener. Comput. Syst.* 17(6), 745–754 (2001)
5. Camp, N., Cofer, H., Gomperts, R.: High-throughput BLAST, <http://www.sgi.com/industries/sciences/chembio/resources/papers/HTBlast/HT.Whitepaper.html>
6. Cardellini, V., Casalicchio, E., Colajanni, M., Yu, P.: The state of the art in locally distributed web-server systems. *ACM Computing Surveys* 34(2) (2002)
7. Chi, E., Shoop, E., Carlis, J., Retzel, E., Riedl, J.: Efficiency of shared-memory multiprocessors for a genetic sequence similarity search algorithm. Technical Report TR97-005, University of Minnesota, Computer Science Department (1997)
8. Dandamudi, S., Yu, H.: Performance of adaptive space sharing processor allocation policies for distributed-memory multicomputers. *JPDC* 58(1) (1999)
9. Darling, A., Carey, L., Feng, W.: The design, implementation, and evaluation of mpiBLAST. In: Proceedings of the ClusterWorld Conference and Expo, in conjunction with The HPC Revolution (2003)
10. Feitelson, D.: A survey of scheduling in multiprogrammed parallel systems. Technical Report IBM/RC 19790(87657) (1994)
11. Gardner, M., Feng, W., Archuleta, J., Lin, H., Ma, X.: Parallel genomic sequence-searching on an ad-hoc grid: Experiences, lessons learned, and implications. In: Löwe, W., Südholt, M. (eds.) SC 2006. LNCS, vol. 4089. Springer, Heidelberg (2006)
12. Grant, J., Dunbrack Jr., R., Manion, F., Ochs, M.: BeoBLAST: distributed BLAST and PSI-BLAST on a Beowulf cluster. *Bioinformatics* 18(5) (2002)

13. Lin, H., Ma, X., Chandramohan, P., Geist, A., Samatova, N.: Efficient data access for parallel BLAST. In: IPDPS, Washington, DC, USA (2005)
14. Lin, H., Ma, X., Li, J., T, Y., Samatova, N.: Processor and data scheduling for online parallel sequence database servers. Technical Report TR-2007-23. North Carolina State Univeristy (2007)
15. Mathog, D.: Parallel BLAST on split databases. *Bioinformatics* 19(14) (2003)
16. McGinnis, S., Madden, T.: BLAST: at the core of a powerful and diverse set of sequence analysis tools. In: *Nucleic Acids Res.* (2004)
17. Message Passing Interface Forum. MPI: Message-Passing Interface Standard (1995)
18. Mu'alem, A., Feitelson, D.: Utilization, predictability, workloads, and user runtime estimates in scheduling the ibm sp2 with backfilling. In: *IEEE TPDS*, vol. 12 (2001)
19. Ostell, J.: Databases of discovery. *ACM Queue* 3(3) (2005)
20. Pai, V., Aron, M., Banga, G., Svendsen, M., Druschel, P., Zwaenepoel, W., Nahum, E.: Locality-aware request distribution in cluster-based network servers. In: *ASPLOS-VIII* (1998)
21. Rosti, E., Smirni, E., Dowdy, L.W., Serazzi, G., Carlson, B.M.: Robust partitioning policies of multiprocessor systems. *Perform. Eval.* 19(2-3), 141–165 (1994)
22. Wang, C., Alqaralleh, B., Zhou, B., Till, M., Zomaya, A.: A BLAST service built on data indexed overlay network. *e-science* (2005)
23. Wang, J., Mu, Q.: Soap-HT-BLAST: high throughput BLAST based on Web services. *BIOINFORMATICS -OXFORD-* (2003)
24. Zhu, H., Smith, B., Yang, T.: Scheduling optimization for resource-intensive web requests on server clusters. In: *SPAA* (1999)