Parallel BLAST Analysis and Performance Evaluation

Juliana C. Correa,
Computer Science Department
UFRJ
Rio de Janeiro, Brazil
jucorrea@ufrj.br

Gabriel P. Silva
Computer Science Department
UFRJ
Rio de Janeiro, Brazil
gabriel.silva@ufrj.br

Abstract

BLAST is an efficient heuristic algorithm used for comparing biological sequences, such as amino-acids or nucleotides and identifying similarities between them. However, extensive genetic mapping projects feed sequence databases continuously, forcing them to grow steadily, turning execution time into a potential constraint to sequential implementations of the algorithm and driving the search for new approaches, such as parallelism, to reduce it.

In this work, parallel implementations of BLAST are evaluated, in particular mpiBLAST, aiming to find the best conditions for executing those applications in high performance parallel systems. Through comparative tests and a time profile of the application, we analyzed its performance variation regarding parameters such as load balancing, advanced load of database fragments in RAM and also employing a high throughput parallel file system.

Keywords: BLAST, MPI, parallel programming, performance evaluation, bioinformatics, genetic database

1 INTRODUCTION

Alignment of proteins or DNA strands, computationally represented as character sequences, is a frequent task in bioinformatics. Exposing the similarities between these sequences may help in the inference of biological functions of recently mapped genes [1] or even in establishing the existence of common ancestors to species.

BLAST - Basic Local Alignment Search Tool [2] – is currently the most adopted algorithm for sequence alignment [5]. A large number of public protein and DNA databases are available, the largest collection of which is stored in NCBI – National Center for Bioinformatics [6], which also provides a popular implementation of the algorithm.

However, the exponential growth of these databases [7] is driving the exploration of techniques that can improve its performance, such as employing parallelism to break the computationally intensive search task into fragments that can be distributed among nodes of a large cluster, achieving the same results in only a fraction of the original time.

This paper evaluates one of the most adopted parallel implementations of the BLAST algorithm, mpiBLAST [3], testing its performance under specific execution conditions and searching for possibilities for its improvement. With this intention, Section 2 further describes BLAST and some of the strategies already employed in its parallelization. Next, Section 3 presents the environment in which the tests were run, followed by their results for different scenarios in Section 4. Finally, Section 5 shows conclusions and some suggestions for future work based on these experiments.

2 BLAST

The first algorithms designed for sequence alignment were Smith-Waterman(1981) and Needleman-Wunsch (1979) [1]. Both are very accurate dynamic programming solutions. However, their complexity is of order O(*nm*), where *n* is the length of the sequence and *m* the length of the database, both representing the number of base pairs, or characters. With the need of matching the query sequence against an entire database, it has become increasingly difficult to find results with these algorithms with the rapid growth of public databases [7].

BLAST, on the other hand, was published in 1990 [2] as an alternative sequence alignment algorithm that employs an approximation algorithm, allowing results to be found in considerably less time, and thus ensuring viability of searches even in large databases. The critical factor for performance improvements with BLAST is a limitation in its search space. Its heuristic approach reduces the initial search to a set of potential candidates and extends the search from them. Nevertheless, by limiting the search space, it may miss some potential alignments, compromising its sensitivity.

Because of inherent differences between nucleotide sequences composing DNA and RNA and amino-acid sequences composing proteins, there are slight variations of the BLAST algorithm for each type of search [2]. The most computationally intensive form of BLAST translates nucleotides sequences to amino-acids sequences created from it both in the database and in the queries and aligns these translated sequences, resulting in a broader range of results. This application, called t-blastx, is used in the tests described in Section 4.

BLAST's final result consists of a series of local alignments, ordered by the similarity score they achieved in the extension process. It presents the complete sequences matched, score, size of the alignment and also a value based on the probability that the alignment has been found by chance, the e-value [2]. The e-value depends on the query and database composition, and some of the alignments with higher e-values are discarded from the result.

Databases in BLAST are in fact text files in FASTA format. FASTA is a regular text file composed of one or more character sequences. Before searching one of these databases, BLAST requires it to be processed by a program called formatdb. This program generates an index and a header file, in order to handle it more efficiently, and also

compresses the sequences, being able to achieve a 4:1 compression rate for nucleotide databases [8]. Albeit query files are in FASTA format as well, they do not need to go through this previous processing - since they tend to be shorter, the procedure of indexing and compressing happens during execution.

2.2 Parallel Implementations

Even though BLAST is able to achieve more performance in time than the previous dynamic programming algorithms, its execution time can still be overwhelming for queries with higher database demands such as nt, the complete nucleotide collection of NCBI, currently reaching 30 GB length in characters.

Because of that, there have been efforts in the area of parallel sequence search, aiming to further reduce the time cost associated to this task. There are three main techniques for that – hardware parallelization, database segmentation and query segmentation.

Hardware parallelization is achieved mainly during the alignment stage, taking advantage of the representation of the search space as a matrix to make the comparison via hardware. BioScan [10] and Tera-BLAST [9] are examples of hardware parallelization, employing VLSI (Very Large Scale Integrated) circuits and FPGA (Field Programmable Gate Array), respectively.

Despite their high efficiency [9, 10], these solutions depend on custom hardware that can be very expensive and require specialized maintenance. Therefore, they are not widely employed because of the costs involved.

On the other hand, database and query segmentation are both software approaches based on data parallelism. That means the core code for BLAST remains the same, as a black box, and it is repeatedly executed in different processors, each of them with a portion of the data. With this technique, code maintenance is easier and cleaner.

Query segmentation is a straightforward method that consists in replicating the database in every node and having each of them search one or more sequences of the query. The implementation is simple, and results can be plainly concatenated, since there is no ordering or dependency between them.

The natural disadvantage is that the query file is required to have enough number of sequences, and load balancing is hard to achieve, since the sequences vary in size and there may not be enough of them to compensate for that. Also, the replication of the database means that instead of distributing data, it requires more space than the sequential method.

In database segmentation, portions of the database are distributed across the nodes. A master-slave model is generally implemented for this purpose, with the slaves performing searches while the master distributes data and receives results.

The greatest advantage is that the whole aggregated memory is used, and it is possible [3] that the database may fit entirely in the distributed primary memory of the nodes, reducing the cost associated with swapping data between RAM and hard disk.

However, after all slaves have finished their search tasks, results must be merged and filtered, according to the e-value. Since these calculations require information about the whole database composition, and slaves only know portions of it, this is usually a task performed by the master [4].

Most available implementations of parallel versions of BLAST are based on this approach. Parallel BLAST [11] runs on PVM environment and claims 35 to 50% efficiency in parallelization for amino-acid alignments and superlinear speed-up for nucleotide searches. Nevertheless, it is noted [3] that it may lack a more explicit load balancing mechanism. TurboBLAST is another implementation, and, even though it features a better load balancing strategy and is dynamically adaptable to the cluster environment, it is a closed proprietary solution.

NCBI BLAST itself has an option to run on an SMP environment through multithreading. Since it is shared memory oriented, this solution does not incur in any overhead when fragmenting the database, though it also does not have the advantage of more primary memory available.

2.3 mpiBLAST

MpiBLAST [3] is a frequently adopted solution for parallelism in BLAST. It was originally based on database segmentation, but has become a hybrid implementation, offering also query segmentation since version 1.4.

Using MPI (Message Passing Interface), it is well adapted to distributed memory environments such as clusters and grids. It accesses core NCBI BLAST code through an interface library called NCBI-Toolbox. That allows both the parallelization strategy and the core algorithm to be updated without interfering with each other.

When running mpiBLAST, the fragments must have been previously created and formatted. However, with this static fragmentation method, the number of fragments must be previously known. A separate application, mpiformatdb, is responsible for that. Not only it partitions the database, it also includes a wrapper for formatdb, generating distributed header and index files. Load balancing depends on an even distribution of sequences among these fragments.

In distributed memory environments, it is usual for mpiformatdb created fragments to reside in a remote storage system. In that case, mpiBLAST loads these fragments to local disks, and, if possible, to core memory. To achieve better performance, it saves these fragments in local disks after the execution, so that a future run may begin with them already loaded and avoid this overhead.

When mpiBLAST starts, the master assigns a fragment to each of the workers. Whenever one of the slaves completes the task and reports to be idle, the master assigns a new fragment to it, according to a heuristic that avoids copies of fragments from the remote storage whenever possible, trying to use previously loaded fragments.

Special routines intercept core BLAST alignment results generated by the slaves, which are then parsed, filtered and sorted by the master, which informs the slaves which results were accepted and asks them to fetch the sequences related to these results. Since version 1.5, there are groups of slaves

implemented on top of the traditional master-slave schema [12], a modification that allows scalability even on massively parallel environments.

Since the master is responsible for consolidation and writing of results, there is a potential serialization constraint in this phase of the process. This is supposedly the main source of overhead in mpiBLAST [4]. Because of that, some improvements have been made in the results processing stage since the original release of the application.

Heshan Lin, Xiaosong Ma and Praveen Chandramohan proposed pioBLAST [4], a set of improvements aiming to minimize costs not directly related to search in mpiBLAST, reducing the non-parallel portion of the program. For that, pioBLAST uses MPI-I/O, an extension o the MPI2 standard that supports parallel I/O and collective access routines.

One of pioBLAST's main updates was the caching of sequences by the slaves as they find potential alignments in their partial results. With this strategy, when informed of which results were accepted, the slaves do not need to access the database again to fetch these sequences.

Besides, pioBLAST also adds dynamic partitioning and parallel output. During partitioning, the master process calculates file ranges for each partition and distributes them to the worker processes, in the form of offset pairs. The worker processes then use these ranges to read, in parallel, different segments of the global sequence and index files into their local memory using MPI-IO.

As for the output, the master sends messages to the slaves informing which alignments were selected and their offsets in the output file. Slaves may then create "views" of the file and write in parallel using collective output routines.

The results of pioBLAST show that it greatly improved performance[4], particularly when a large number of processes are involved or the output file is large. Even though search time increases proportionally in both programs, with 62 processes, it is shown that pioBLAST spends 92.4% of its time in the search, while mpiBLAST spends only 10.3%, not being able to show any scalability in such situation.

Because of these results, some of pioBLAST's enhancements were added to mpiBLAST, resulting in the development of mpiBLAST-pio, which is the official version of mpiBLAST since release 1.6. One of its main contributions is the caching of complete sequences in the slaves during search.

Parallel output was also adopted, though with some differences - while in pioBLAST results are buffered to generate large blocks and take advantages of collective writing, in mpiBLAST-pio, this is not possible, because results are not buffered, to avoid complications with large queries. Therefore, mpiBLAST also keeps non-collective routines. However, such routines have their performance degraded in non-parallel systems, such as NFS. For that reason, mpiBLAST includes two output options, **-use-master-write** and **–use-parallel-write**.

MpiBLAST does not adopt dynamic partitioning. The reasons for that are mainly the lack of support to large files [4] and the possibility of keeping the fragments loaded in local disks when using static partitioning.

3 TEST ENVIRONMENT

3.1 Cluster

All the experiments were run in a HPC cluster installed in NCE/UFRJ. According to HPL benchmark suite, Netuno, as the cluster is called, can sustain 16.2 Teraflops [14], having reached position 138 in the Top500 list of the fastest supercomputers in the world in June 2008, when it was installed.

The cluster is composed of 256 computing nodes, each of them with two quadcore Intel E-5430 processors, 12 MB cache, 16 GB RAM, meaning there are 2048 cores available for processing. Each node contains a 160 GB hard disk, in which the operating system and a temporary area for user jobs are available.

The intercommunication network between the computing nodes consists in a full-duplex 20 Gbps Infiniband network, allowing for highly efficient message passing. Another network is setup for I/O and management tasks, linking each rack of nodes through 2x10 Gbps fiber to a Cisco 6509 core switch, which is, in its turn, connected to the storage systems through 12 Gigabit Ethernet channels. In the following tests, the Gigabit Ethernet network is used for both I/O and intercommunication.

There are three MPI implementations available in the cluster – OpenMPI, Intel MPI and MPICH. Among these, OpenMPI was chosen for these tests for being an open and robust implementation. With it, mpiBLAST was compiled from version 1.5, current version at the time of the tests, available from mpiBLAST website [16] and including the improvements of pioBLAST. NCBI BLAST was compiled from version 2.2.21, available from NCBI website [6].

3.2 File systems

File systems play an important role in the performance of mpiBLAST. Both the input reading of the database and the writing of the output file are potential causes of overhead, not being part of the core parallel function of searching.

Netuno includes both a parallel file system and a NFS file system. The first one is a scratch system designed for input and output at runtime, while the second is a more permanent storage keeping home directories and users' files for future consulting.

The parallel file system, PanFS [17], is a system developed by Panasas, Inc., which aims for high levels of performance and stability. In order to achieve that, it uses dedicated hardware, holding a total of 30 TB storage capacity. A protocol called Direct Flow (pNFS), responsible for coordinating the access of multiple nodes, implements object orientation, allowing compute nodes to communicate directly to storage nodes, without intervention of a metadata server, as is the case in NFS file systems. In preliminary tests [15], it was observed that it can reach a total throughput of 422 MB/s, or 3.3 Gbps, about 1/3 of the network capacity.

3.3 Data

NCBI databases are public and contain most genome and proteins already sequenced. Among nucleotide bases, used by tblastx, the largest one is the nt database, with 30 GB distributed in over 11 million sequences, at the time. The query files were randomly sampled out of the nt database, ranging from sizes of 11 KB in 3 sequences up to 1 MB in 1442 sequences.

Since our cluster has 16 GB available memory for each node with eight cores, and it is recommended to have up to 80% RAM free to the database fragments [4], they should be kept to a maximum of 1.6 GB in order to keep the entire database in aggregated memory, as mpiBLAST expects. Despite being 30 GB in size, after being indexed and compressed, the nt database has about 16 GB. With that, it would be recommended to execute in a minimum of two nodes, meaning 16 processes, or 14 fragments.

Mpiformatdb also imposes limitations to the minimum and maximum fragments. It is able to generate at most 250 fragments, and fails in the creation of fragments larger than 2 GB. Taking compression into account, it would not be possible to split the nt database in less than 10 fragments.

All tests performed were run at least five times, and the minimum and maximum measures were discarded. The final value is the average of the remaining measures.

4 RESULTS

4.1 Efficiency

As a general goal, mpiBLAST's general performance is evaluated in this test, measuring speed-up and efficiency in comparison with the sequential NCBI BLAST version. Since NCBI BLAST reads data from the local disk, there were preliminary runs in mpiBLAST so that the fragments were previously loaded to local disks as well.

Figure 1 shows the speed-up obtained versus the number of processors, compared to the straight line that represents linear speed-up or 100% efficiency.

It is possible to observe that up to 128 processes (16 nodes) the speed-up keeps above linear, in accordance with the results presented in [3, 4]. Beyond that point, despite it is no longer superlinear, efficiency is still not far from 100%, demonstrating good scalability results for mpiBLAST.

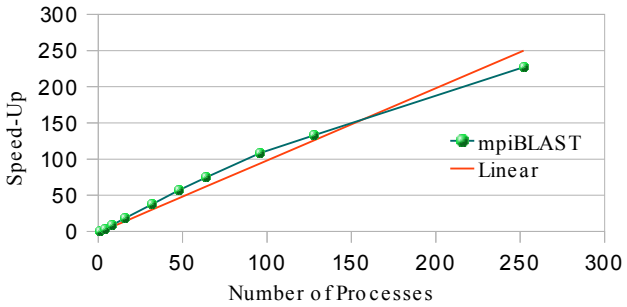

Figure 1. MpiBLAST speed-up

Sequential time for the 11 KB query, the smallest of the queries used was of 4h 34min, while with 252 processes, it reached 76 seconds, a 216 times speed-up, meaning 86% efficiency. With 128 processes the same query took 123 seconds to produce results, with 138 times of speed-up representing 108% efficiency.

4.2 MpiBLAST and Multithreaded NCBI BLAST

As observed in Section 2.2, NCBI BLAST offers the possibility of multithreading in SMP systems. This test aims to compare mpiBLAST's performance with this option.

Despite the parallelism in the search, NCBI can be run in only one node, thus for comparison mpiBLAST is also executed in only one node, so that both have the same available memory. Since NCBI BLAST runs from local hard disk, previous executions of mpiBLAST were also run in order to start its execution with fragments preloaded.

Table I shows that NCBI BLAST keeps a regular linear speed-up behavior, which is a positive measure of efficiency. However, since mpiBLAST shows superlinear speed-up, this test demonstrates that even in a system with only one multicore node it is preferable to use mpiBLAST instead of trying the multithreaded NCBI solution.

TABLE I. MPIBLAST VS MULTITHREADED NCBI BLAST

program	Number of threads/slaves			
	2	*4*	*6*	*8*
mpiBLAST	5292 s	3598 s	1926 s	1965 s
NCBI	8100 s	4090 s	2705 s	2059 s

The main reason for mpiBLAST's superior performance is that while NCBI BLAST implements only a parallel search, mpiBLAST adds other improvements, such as the caching of sequences and the results merging phase.

It is worth noting that this is a comparison of the total number of threads against the total number of slave processes, since the two other mpiBLAST processes, master and super master, do not participate in the parallel search.

4.3 Load Balancing

The key to load balancing in mpiBLAST lies in the distribution of sequences among the slaves. It is important that all the slaves take approximately the same time in searches, so that no process is idle for too long.

During partition, some fragments may have more characters than others, or even they have same number of characters irregularly distributed, some of them with a small number of large sequences and others with a large number of small sequences. It is also possible that unbalanced situations are caused by similarity between the query and the sequences, with a large number of hits to be extended in some of the fragments.

Nevertheless, even with balanced fragments and hits statistically distributed, there still may be load unbalancing if these fragments are not properly distributed among the

slaves. In the next two tests, we wish to observe the behavior of mpiBLAST regarding the distribution of its fragments, analyzing the effects of different rates between the number of processes and the number of fragments.

In the first experiment, there are 14 slaves involved in the search and the number of fragments was raised progressively from 14 to 150. With that, the number of fragments searched by each process ranges from only 1 up to 17.

Figure 2 shows that, as expected, natural partitioning, where the number of fragments is an integer multiple of the number of slaves, is the most adequate option, with performance 20 to 30% better than other ways of partitioning. When the number of fragments is equal to 30, 62 or 250, the processes do not receive the same number of fragments. In those cases there is a load unbalance, and performance is degraded. Because of this, there are local minimums shown in the graphic for 14 and 126 fragments, when balance happens naturally.

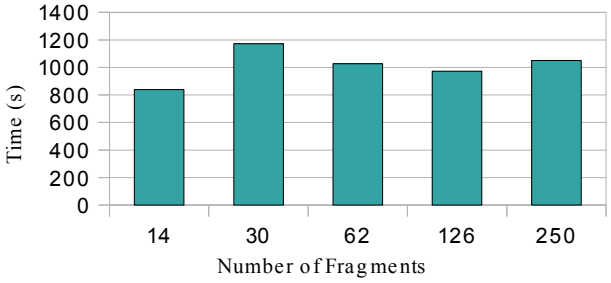

Figure 2. MpiBLAST's performance with 14 slaves

In the second test, the number of fragments is fixed to 250, and the number of slaves is progressively reduced from 125 to 2. The goal is comparing the performance of fixed partitioning to that of natural partitioning.

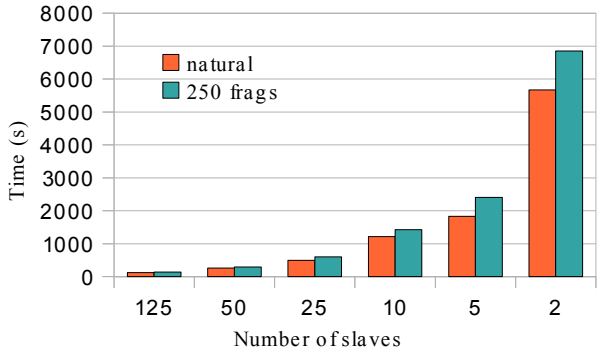

Figure 3: MpiBLAST's performance with 250 fixed fragments

Figure 3 shows that the overhead of fixed partitioning, compared to natural partitioning, grows proportionally with the number of fragments searched by each slave. Possibly, the main reason for that is the increase in complexity of merging partial results.

This experiment shows that, despite it is possible to create only one partitioning with the highest possible number of fragments and use it with any number of processes, it implies that mpiBLAST will not reach its best performance. Therefore, this approach is not advisable.

4.4 Data Scalability

This test tries to evaluate BLAST's behavior as the query file and the output file increase. The term data scalability was used in [4], with similar tests which showed that mpiBLAST had a limitation in scalability because even though the parallel search could scale well, the result consolidation phase could not.

In Figure 4 it is possible to notice that as the query file increases in size, execution time increases in greater proportion, to the point of growing seven times between the 350 KB and 1 MB queries. Even though the 1MB query is only 100 times greater than the 11KB query, the execution time grows 411 times when using 128 processes and 294 times when using 250.

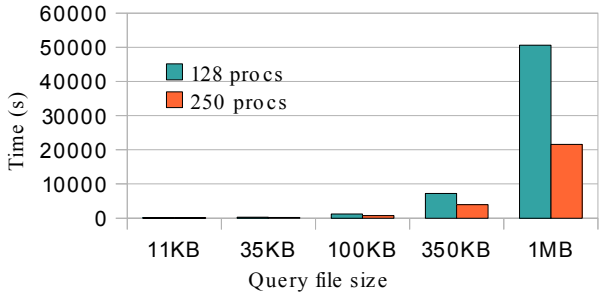

Figure 4: Data Scalability

One of the reasons for that is that the process of ordering and filtering the results becomes more costly and complex, as does the writing in the output file, which increases considerably, ranging from 6.3 MB in the 11 KB query to 3 GB in the 1 MB query – 500 times larger, even though the query itself is only 100 times larger.

It is possible to conclude that data scalability is limited, increasing cost in proportions much larger than the increase in the query file itself. However, for medium sized and most frequent queries, it might be an acceptable cost.

4.5 Parallel write (PanFS) x Concurrent write (NFS)

The goal of this test is to compare the writing performance of mpiBLAST in a high performance parallel file system – PanFS – against an ordinary NFS remote file system. MpiBLAST allows to switch between parallel and master write. When parallel write is activated, slaves are responsible for writing the output file, in parallel, in the offsets designated by the master. Otherwise, the master process receives from the slaves the sequence data they had

cached and writes the file sequentially. The first option is indicated for PanFS, while the second one for NFS.

Figure 5 shows that for a small query file, 100 KB, parallel write does improve performance over NFS, yet this improvement is not so significant, representing a reduction of 5% to 9% in execution time.

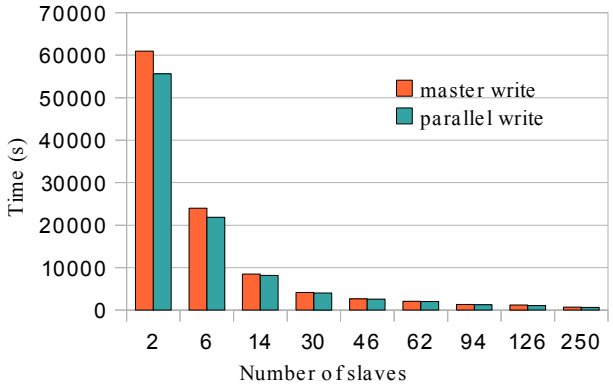

Figure 5: Parallel and master write for 100 KB query file

This behavior reflects the considerations of pioBLAST's authors [4], about the overhead being caused not because of the file writing itself, but by the serialization in the results consolidation phase, where the master was the only process responsible for filtering, ordering and also fetching the sequence to be written for each result.

On the other hand, when the test is run over larger queries, results indicate a more significant performance gain with the use of PanFS, reaching about 15% gain with 32 processes, as can be observed in Figure 6, possibly because of the advantage of distributing the writing of a much larger output file. However, as Figure 6 also shows, when the number of slaves increases, there is an overhead probably introduced by the large number of processes trying to access the same file for simultaneous writing, and the gain is once more reduced, though parallel write is still faster than master write.

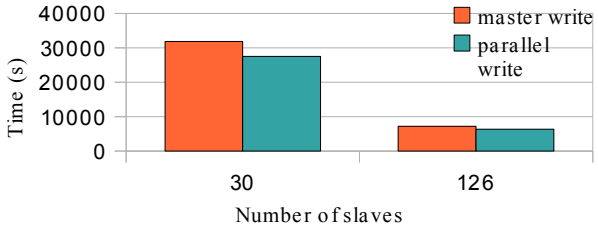

Figure 6: Parallel and master write for 1 MB query

With that, it is possible to conclude that when a parallel file system is available, it is in general worth activating the parallel write option, specially for alignments with large query sizes and not too many slaves.

4.6 Reading and loading database fragments

This test is aimed at evaluating the impact of reading the database and loading its fragments from remote storage. For the first test, the local disk was cleaned and two consecutive runs of mpiBLAST were fired. Another situation was tested, simulating execution in a diskless cluster.

These tests were repeated in PanFS and NFS. Unlike last section, where the goal was to measure the impact of parallel output, this tests compare parallel input to regular input.

The results are presented in Figure 7. In the first column are the executing times when reading the fragments from local disk, excluding the time of the copy itself. As expected, there is no difference between PanFS and NFS, since the remote file system is not involved.

In the second column, where the time to copy the fragments from the remote system to local disk is included, it can be noted that there is a significant improvement using PanFS. The total execution time is reduced from 819 s to 317 s. This shows that the copy process can automatically employ the Direct Flow protocol, taking advantage of parallelism in simultaneous transfer of fragments to different nodes. Finally, the third column show the times when fragments are read straight from the remote storage, where again PanFS has a significant advantage over NFS.

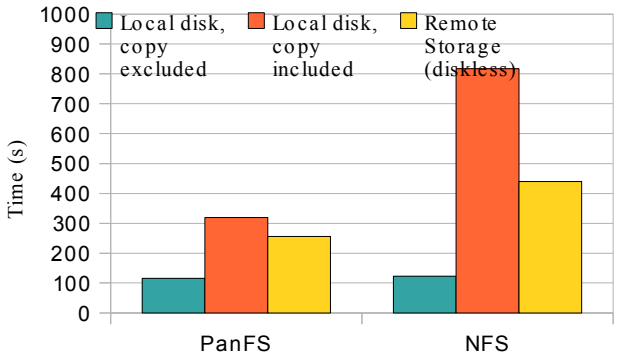

Figure 7: Database reading comparison

Another fact that stands out from Figure 7 is that the total time for executing the program from remote storage is considerably less than the time to copy the fragments and execute from the local disk. If the user cannot make sure that the next run will be granted the same nodes as the current one, this indicates that it is better to execute straight from remote storage instead of allowing the copy of fragments to the local disk.

4.7 Time Profile

For a better comprehension of the application's behavior, a time profile of it was traced. Two sources were used - data logged in mpiBLAST debug mode and calls added in the code with MPI_WTime, MPI library function used to record elapsed time. The application was split into the following stages:

1 – Preparation – initialization of processes and MPI

2 – Copy – Maximum time among nodes to copy the fragments from remote storage to local disks.

3 – Blast – Maximum core search time

4 – Results Processing – time the master spends waiting for partial results, merging, filtering, calculating offsets and optionally sending them to slaves.

5 – Write – amount of time spent in write and waiting write. It may represent the absolute time of the master (in master write) or the maximum time among slaves (in parallel write).

In Figures 8, 9 and 10, it is possible to see this profile applied to different situations. Preparation time was omitted because it took less than 2 seconds in all tests.

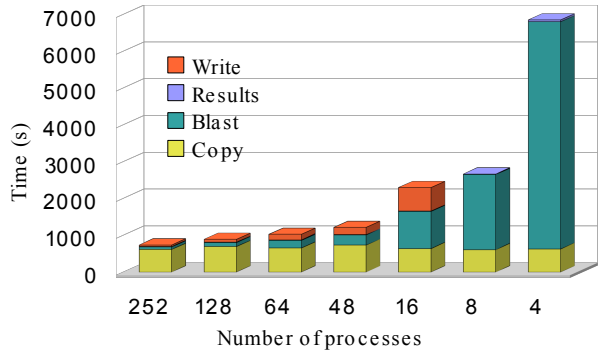

Figure 8: Time profile of 11 KB query in NFS

Figure 8 shows an 11 KB query over NFS using master write. It shows that search (blast) time increases in the inverse reason of the number of processes involved. That portion of the program is the one that gains more benefits from parallelization. With less than 8 processes, the write time is negligible, since there is only one node involved and the result file is small. In the other cases there is the need to coordinate the writing of the many nodes involved in the search over the network. The copy time, however, does not show any variation, since it depends mainly of network and storage throughput, which are the same in all cases.

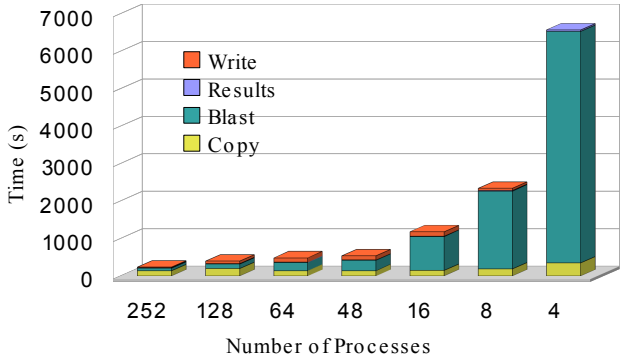

Figure 9: Time profile for 11 KB query on PanFS

With more than 16 processes, the results processing time, despite it is very small, grows as more processes are involved, expressing the increase in the complexity of filtering and merging operations.

Figure 9 shows the same query run over PanFS using parallel write. Copy time decreases very much (2 to 3 times), as shown also in Section 4.6, due to the use of a parallel file system. The write time also shows the some reduction with more than 8 processes, although not significantly as the copy time. The blast and the results processing stage times are practically identical to the ones using NFS, as expected.

Next, Figure 10 corresponds to the execution of a query of 100 KB over NFS. There is only one sequence in the query, opposed to the last query, which was composed of three sequences. With that, there is a reduction in the results processing time (3 to 4 times), which represents a very small fraction of this profile, even though the output file itself is bigger than the previous one. This is an indication that the complexity of the offset calculation should be increasing when there are multiple sequences in the query, degrading performance of the writing phase.

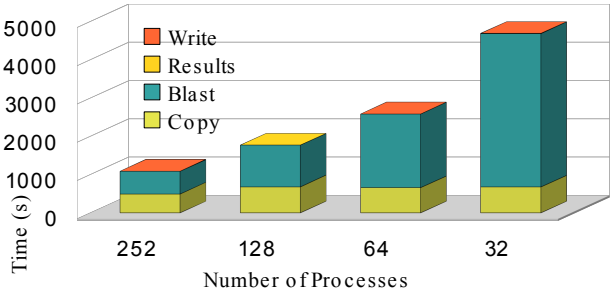

Figure 10: Time profile for 100 KB query in NFS

5 CONCLUSIONS AND FUTURE WORK

5.1 Test Conclusions

Throughout the tests, we analyzed mpiBLAST, exposing its adaptation to different circumstances, particularly for HPC clusters. It was observed, by means of comparative tests and of a time profile, that the parallelization is efficient and scalable, spending the most of its time in its main task, which is the search of alignments, specially after pioBLAST's additions.

It was also noticed that for the best performance, it is highly recommended that the number of fragments is the exact same number of slaves performing the search, or, when that is not possible, that the number of fragments is an exact multiple of the number of slaves, thus avoiding load unbalance, which would greatly degrade its performance.

Whenever a parallel file system is available, it is also recommended that the option **–use-parallel-write** is activated, because it always provides performance gain, specially in larger queries and with a smaller number of slaves involved.

Even if there is only one multicore node available, mpiBLAST should be preferred to NCBI's multithreaded solution, since its optimizations grant it a better performance than the latter.

With the time profiling, it was determined that query files with more sequences tend to incur in more overhead in results consolidation and writing, being adequate to break them into smaller portions.

Finally, it was shown that the results phase, previously [3,4] pointed out as the greatest cause of overhead was consistently improved after pioBLAST's efforts, through caching of sequences in slaves and transference of fetching responsibility to them.

5.2 Suggestions and future work

Throughout this work, it was shown that static partitioning has a series of drawbacks, specially in an environment where available resources might vary along time, as in a cluster.

The first one is that the need to keep a fixed number of fragments leaves the user with the choice of either keeping a fixed number of processes as well, not being able to fully take advantage when more nodes are available, or varying the number of processes without, however, being able to achieve full performance. In large queries, this overhead may represent some extra hours to execution time. Even though it is possible to generate several partitioning schemes, it would not solve the problem entirely, since it would not be reasonable to generate all possible schemes, and also it would consume too much space in disk.

Besides, tests in Section 4.6 demonstrate an important fact – that time spent copying the fragments to local disks before execution surpasses time spent if the program is executed reading the database straight from the remote system, as if it was in a diskless cluster. In a cluster, where job management systems may allocate resources to other programs between runs of mpiBLAST, it is not always possible to ensure that the same nodes will be available for consecutive rounds. Therefore, it is possible that the user subsequently incurs in the overhead of copying the fragments to different disks.

Given these circumstances, it would be possible to consider the adoption of partitioning during runtime, generating fragments that always match exactly he number of slaves available. Currently, that is not viable, since it would mean an spending about the same time of mpiformatdb, which means about one hour, as overhead to each run of mpiBLAST. However, mpiformatdb is a sequential tool probably running over the parallel environment used for mpiBLAST, and it could be a possibility to make it a parallel tool embedded in mpiBLAST.

As a collateral effect, there would be no need to allocate an extra process for the group master, allowing for one more core to works as a slave. This could make a difference in systems with few cores available, as observed in Section 4.2.

Finally, it should be carefully considered that since the search achieves superlinear speed-up and the overhead in result merging seems to be settled, there may be more room for improvement either in the input phase or in the core logic of BLAST. One of mpiBLAST's greatest vulnerabilities is, as a matter of fact, inherited from NCBI BLAST – the maintenance of databases in text format. Besides not providing the established and developed mechanisms to search optimizing present in traditional DBMS, being a text format, FASTA does not have the appropriate means to ensure the uniqueness of the sequences. On top of that, they are harder to maintain and update, either to apply patches with the updated NCBI versions or to add local sequences mapped by the end users.

For all that, it seems possible that a proper managed database would improve performance, maintainability and ease of use of both NCBI BLAST and mpiBLAST.

6 REFERENCES

[1] David W. Mount, "Bioinformatics: Sequence and genome analysis," Cold Spring Harbor Laboratory Press, 2004.

[2] S. Altschul, W. Gish, W. Miller, E. Meyers, D. Lipman, "Basic Local Alignment Search Tool**.,"** Journal of Molecular Biology, 215(3), 1990.

[3] Aaron E. Darling, Lucas Carey, Wu-chun Feng, **"**The Design, Implementation, and Evaluation of mpiBLAST**,"** 4th International Conference on Linux Clusters, 2003.

[4] Heshan Lin, Xiaosong Ma, Praveen Chandramohan, "Efficient Data Access for Parallel BLAST," 19th International Parallel & Distributed Processing Symposium, 2005.

[5] David Wheeler, Medha Bhagwat. "Comparative genomics," Humana Press, 2008.

[6] National Center for Bioinformatics website, http://www.ncbi.nlm.nih.gov.

[7] GenBank growth statistics, revised in February 2009, http://www.ncbi.nlm.nih.gov/genbank/genbankstats.html .

[8] FormatDB documentation, http://www.ncbi.nlm.nih.gov/BLAST/docs/formatdb.html.

[9] TimeLogic, "Tera-BLAST algorithm module", available in http://timelogic.com/downloads/ TeraBLAST_2009.pdf.

[10] Raj K. Singh, W.D. Detloff, V.L. Chi, " BioScan: A Dynamically Reconfigurable Systolic Array for Biosequence Analysis," CERCS96, National Science Foundation, 1996.

[11] David R. Mathog, "Parallel BLAST on Split Databases," Bioinformatics Applications Note Vol. 19 , pg 1865-1866, 2003.

[12] Oystein Thorsen, Brian Smith, Carlos Sosa, Karl Jiang, Heshan Lin, Amanda Peters, Wu-chin Feng, "Parallel Genomic Sequence Search on a Massively Parallel Environment," 4th International Conference on Computing Frontiers, 2007.

[13] Christopher Joseph Goddard, " Analysis and Abstraction of Parallel Sequence Search," Msc. Thesis, Virginia Polytechnic Institute and State University.

[14] Vinicius Silva, Cristiana Bentes, Sergio Guedes, Gabriel P. Silva, " Arquitetura e Avaliação do Cluster de Alto Desempenho Netuno," Workshop em Computação de Alto Desempenho, 2009.

[15] Juliana Correa, Gabriel P. Silva, "Avaliação de Desempenho do Sistema PanFS Sobre o Cluster Netuno", Workshop em Computação de Alto Desempenho, Iniciação Científica, 2009.

[16] mpiBLAST website, http://www.mpiblast.org

[17] http://www.panasas.com/products/panfs.php