A Multiple Task Allocation Framework for Biological Sequence Comparison in a Grid Environment

Azzedine Boukerche[1], Marcelo S. Sousa[2] and
Alba C. M. A. de Melo[2]

[1]University of Ottawa
School of Information Tech. and Engineering
800 King Edward Avenue Ottawa, Canada
boukerch@site.uottawa.ca

[2]University of Brasilia (UnB)
Dept. of Computer Science
ICC-Norte Subsolo, Brasilia, 70910-900, Brazil
{albamm,marcelo}@cic.unb.br

Abstract

The evolution of DNA sequencing techniques generated huge sequence repositories and hence the need for efficient algorithms to compare them. To increase search speed, heuristic algorithms like BLAST were developed and are widely used. In order to further reduce BLAST execution time, this paper evaluates an adaptive task allocation framework to perform BLAST searches in a grid environment against segmented genetic databases segments. Our results present very good speedups and also show that no single task allocation strategy is able to achieve the lowest execution times for all scenarios. Also, our results show that the proposed adaptive strategy was able to deal with the heterogeneous and non-dedicated nature of a grid.

1 Introduction

In the last decades, an extremely high number of organisms have been sequenced in genome projects. After determining a protein (or nucleotide) sequence, one must infer its function. Usually, this is done by comparing the newly sequenced organisms against sequences for which the functionality has already been established. Many efforts have been made to collect the annotated sequences and place them in publicly acessible databases. The problem is that these genetic databases are huge and, by now, they present an exponential growth rate.

Therefore, biological sequence comparison, also called sequence alignment, is one of the most important problems in computational biology, given the number and diversity of the sequences and the frequency on which it is needed to be solved daily all over the world [11]. The most important types of sequence alignment problems are global and local. To solve a global alignment problem is to find the similarity between the entire sequences. Local alignment algorithms must find the similarity between parts of the sequences. In this article, we will treat mainly local alignments.

Smith and Waterman [14] proposed an algorithm (SW) which is a variation of the algorithm proposed by Needleman and Wunsh that finds the best local alignment between two genomic sequences. Its time and space complexity is also $O(n^2)$. In genome projects, the size and the number of sequences to be compared are constantly increasing, thus an $O(n^2)$ solution is still expensive. For this reason, heuristics were proposed to reduce the time spent in computation. BLAST [1] is an example of a widely used heuristics to compute local alignments.

The popularity of the Internet made possible the interconnection of millions of powerful machines in a global scale. Several measures were made which stated that, most of the time, the majority of these interconnected machines remain idle. This led to the idea of metacomputing [13], which proposes the creation of a supercomputer by taking advantage of the idle cycles of the machines connected to the Internet.

Grid computing is considered to be an evolution of metacomputing where not only the computing power of the machines is shared, but also several other types of resources such as data, softwares and specific hardwares [4]. Applications which are developed for grid environments are very complex since they have to deal with a great number of heterogeneous and non-dedicated resources placed on multiple administrative domains.

1-4244-0054-6/06/$20.00

Initially, the efforts in grid computing were concentrated to develop middlewares which were able to offer a wide area infrastructure to support the online processing of distributed applications [8]. The Globus Toolkit [3] is considered to be the "de facto" standard for grid middleware, offering basic solutions for problems such as authentication and remote task execution, providing the basic infrastructure where real distributed applications for grid computing can be developed.

Since grid applications are designed to run in a geographically distributed environment, they usually do not have high communication taxes and many of them follow the master/slave model [12]. In order to schedule master/slave applications many task allocation policies were proposed such as *Self Scheduling* [15] and *FAC2* [5]. The choice of the best allocation policy depends on the application access pattern and on the environment in which it runs [12].

In this paper, we evaluate *PackageBLAST*, an adaptive multi-policy framework structured as a grid service to run BLAST searches in a grid environment composed by segmented databases. *PackageBLAST* executes on Globus 3 [3] and, by now, provides five allocation policies. Also, *PackageBLAST* incorporates a generic adaptive mechanism called *Package Weighted Adaptive Self-Scheduling (PSS)*, to assign weights to the grid nodes that takes into account current workload and heterogeneity.

Our results in a heterogeneous grid environment composed by 16 machines show that very good speedups were achieved. When compared with the average machine, we achieved a speedup of 14.59 with 16 machines, reducing BLAST execution time from 30.88 minutes to 2.11 minutes. Also, we show that, in our test environment, there is no single task allocation policy that produces the best execution times for BLAST.

The remainder of this article is organized as follows. In section 2, we present an overview of BLAST. Section 3 describes allocation policies for Master/Slave applications. Section 4 discusses related work in the area of distributed BLAST. Section 5 presents PackageBLAST. Experimental results are discussed in section 6. Finally, section 7 concludes the paper and indicates future work.

2 BLAST Overview

To compare two sequences, we need to find the best alignment between them, which is to place one sequence above the other making clear the correspondence between similar characters [11]. In an alignment, spaces can be inserted in arbitrary locations along the sequences so that they end up with the same size.

Usually, thousands of biological sequences are compared daily against millions of sequences that compose genetic data banks. Due to the current growth rate, these databases will soon achieve terabytes. In this scenario, the use of exact methods is prohibitive. For this reason, faster heuristic methods are proposed which do not guarantee that the best alignment will be produced. Usually, these heuristic methods are evaluated using the concepts of sensitivity and sensibility. Sensitivity is the ability to recognize more distantly related sequences, i. e., it consists of finding *all* real alignments (true positives). Searches with a high sensitivity are more likely to discard false positive matches. Selectivity is the ability to narrow the search in order to retrieve *only* the true positives. Typically, there is a tradeoff between sensitivity and sensibility.

2.1 BLAST Algorithm

BLAST [1] is based on a heuristic algorithm which was designed to run fast while still maintaing high sensibility. It provides programs for comparing many combinations of query and database sequence types by translating sequences on the fly. For instance, BLASTN and BLASTP compare two sequences of nucleotides and proteins, respectively.

The BLAST algorithm is divided into three well-defined phases: seeding, extension and evaluation.

In the first phase, BLAST compares a query sequence s against all sequences t in a database. BLAST uses the concept of words which is defined to be a finite set of letters with length w that appear in a given sequence. For instance, the sequence TCACGA contains four words with length 3: TCA, CAC, ACG and CGA. The BLAST algorithm assumes that significant alignments have words in common.

The location of all shared w-letter words between sequences s and t is determined by doing exact pattern matching. Only regions with identical words are used as seeds for the alignment.

For the cases where significant alignments do not contain words in common, the concept of neighborhood is used. The neighbor of a word includes the word itself and every other word whose score is at least equal to T, when compared through a substitution matrix.

An appropriate choice of w, T and the substitution matrix is an effective way to control the performance and the sensibility of BLAST.

The seeds obtained in the previous phase must be extended in order to generate an alignment. This is done by inspecting the characters near the seed in both directions and concatenating them to the seed until a

drop off score X is reached. The *drop off score* defines how much the score can be reduced, considering the last maximal value. Having the seed A, the X parameter equal to 4 and a punctuation of $+1$ for matches and -1 for mismatches, the following result is obtained:

```
ATGC GATA CTA
ATTC GATC GAT
1212 3454 321 <-- score
0010 0001 234 <-- drop off score
```

After that, the algorithm goes back to the best score (in this case, 5) to obtain the alignment. The alignments generated in the extension phase must be evaluated in order to remove the non-significative ones. The significant alignments, called High Score Segment Pairs (HSPs) are the ones whose scores are higher or equal to a threshold S. Also, consistent HSP groups are generated that include non-overlapped HSPs that are near the same diagonal. The consistent HSP groups are compared against a final threshold, known as the E parameter, and only the alignments that are above this threshold are considered.

3 Task Allocation for Master/Slave Applications

Given a master/slave application composed by a master m and S slaves, the task allocation function $allocate(m, s_i, N, S)$ determines how many tasks out of N must be assigned to a slave s_i (equation 1) [12]. In this equation $A(N, S)$ represents a particular allocation policy. The expression $WeightFactor(m, s_i, S)$ was defined by [12] (equation 2) and provides weights for each slave s_i, based on its processing rate per unit of work.

$$allocate(m, s_i, N, S) = A(N, S) * WeightFactor(m, s_i, S) \quad (1)$$

$$WeightFactor(m, s_i, S) = \frac{P * WorkerRate(m, s_i)}{\sum_{i=1}^{P} WorkerRate(m, s_i)} \quad (2)$$

The function $WorkerRate(m, s_i)$ considers static information previously known from computer nodes and is defined as the work unit completion rate occurring between master m and worker s_i, in units of work per units of time.

The following paragraphs present some work unit allocation policies. Each strategy is an instance $A(N, S)$ of equation 1.

The *Fixed* (Static Scheduling) [12] strategy distributes all work units uniformly to slaves nodes. This strategy is appropriate for homogeneous systems with high communication latencies, whose resources are dedicated (equation 3).

$$A(N, S) = \frac{N}{S} \quad (3)$$

Self-Scheduling (SS) [15] distributes a single work unit to each slave node. This procedure continues until all work units are allocated (equation 4).

$$A(N, S) = 1, \textit{while work units are still left to allocate} \quad (4)$$

In SS, the maximum idle time a set of nodes could wait for is limited by the processing time of a single work unit in the slowest node. Nevertheless, SS often demands a lot of communication.

Trapezoidal Self Scheduling (TSS) [16] allocates work units in groups with a linearly decreasing size. This strategy incorporates two variables, *steps* and δ, that represent the number of allocation steps and the block reduction factor, respectively. Equations 5 and 6 show how these variables are calculated.

$$steps = \left\lceil \frac{4NS}{N + 2S} \right\rceil \quad (5)$$

$$\delta = \frac{N - 2S}{2S(steps - 1)} \quad (6)$$

The TSS allocation function is presented in equation 7. It calculates the length of the s^{th} block using the difference between the length of the first block and total reduction from the last $s - 1$ blocks.

$$A(s, N, S) = max\left(\left\lfloor \frac{N}{2S} - [(s - 1) * \delta] \right\rfloor, 1\right) \quad (7)$$

Guided Self Scheduling (GSS) allocates work units in groups whose length decrease exponentially (equation 8). The main disadvantage is the potential to allocate large chunks to a slow machine, causing an imbalance in final processing time.

$$A(s, N, S) = max\left(\left\lfloor \frac{N\left(1 - \frac{1}{S}\right)^{s-1}}{S} \right\rfloor, 1\right), s \text{ ¿ } 0 \quad (8)$$

Factoring ($FAC2$) allocates work units in groups organized by cycles. Each cycle consists of S allocation sequences (equation 9). In $FAC2$, half of the remaining work units are allocated to active nodes in each allocation round (equation 10). Note that the size of the allocation blocks decrease with an exponential rate of 2. That's why this strategy is called $FAC2$.

$$round(s) = \left\lfloor \frac{(s - 1)}{S} \right\rfloor + 1 \quad (9)$$

$$A(s, N, S) = max\left(\left\lfloor \frac{N}{S * 2^{round(s)}} \right\rfloor, 1\right) \quad (10)$$

4 Related Work

In genome projects, hundreds or even thousands of newly sequenced organisms must be compared against genetic databases using BLAST. In order to accelerate these searches, distributed versions of BLAST have been proposed that run in cluster or grid environments.

MpiBLAST [2] is a parallel tool that runs BLAST in clusters. The algorithm has two phases. First, the genetic database is segmented and put in a shared storage medium. Next, the queries are evenly distributed among the cluster nodes. If the node does not have a database fragment, a local copy is made. When the slaves finish processing, they send the local reports to the master. Having received the local reports for a query sequence, the master merges them to create the final BLAST report.

Parallel BLAST++ [9] uses query packing to group multiple sequences in order to reduce the number of database accesses. A master/slave approach is used that allocates the queries of BLAST++ to the slaves according to the *fixed* policy (section 3).

GridBlast [7] is a master/slave grid application based on BLAST that uses Globus 2. It distributes sequences to be compared (queries) among the grid nodes using two task allocation policies: *First Come First Served* and *minmax*. Of those, only the last one considers the current load and the heterogeneity of the environment. However, to use *minmax*, the total execution time of each BLAST task in each node must be known.

Grid Blast Toolkit (GBTK) [10] offers a framework and a web portal to execute BLAST searches in a Globus 3 environment. All genetic databases that will be used by BLAST are statically placed on the grid nodes (without replication). The BLAST grid service offered by GBTK is a master/slave application that receives the sequences to be compared and the name of the genetic database to be used. It then verifies if the node that contains the database is available. If so, it is selected to do the search. If the node is not available, the less loaded node is chosen and the database is copied to it.

5 Design of *PackageBLAST*

As [12], we think that there are no allocation policy that is able to produce the best results for all scenarios and that heterogeneity must be taken into consideration. Unlike [12], we think that the non-dedicated nature of a grid environment must also be considered when assigning tasks to nodes. This motivated us to create a framework where many task allocation policies can be integrated. The size of the work units assigned is calculated using the current allocation policy and a weight factor that takes into consideration both heterogeneity and current local load. As in MpiBLAST (section 4), the genetic database is split into smaller pieces, in order to reduce search time.

5.1 Database Segmentation and Distribution

Segmentation consists in the division of a database archive in many portions of smaller size (segments), that can be processed independently. Segmentation enables grid nodes to search smaller parts of a sequence database, reducing or even eliminating unnecessary disk accesses and hence improving BLAST performance.

Just as in MpiBLAST (section 4), we decided to use database segmentation technique in *PackageBLAST* with an NCBI tool called *formatdb*, which was modified in order to generate more database segments of smaller size.

We opted to replicate the segmented genetic database (in our case, *nr*, which has around 1.2GB) in every slave grid node to improve data accesses times and to provide a potential for fault tolerance. If a grid node fails, another node could assume the work, since all database segments are replicated in all grid nodes.

5.2 Task Allocation

We propose the use of a framework where many allocation policies can be incorporated. By now, our framework contains five allocation policies: *Fixed*, *SS*, *GSS*, *TSS* and *FAC2*, all described in section 3. So, the user can choose or even create the allocation policy which is more appropriate to his/her environment.

Besides that, we propose *PSS (Package Weighted Adaptive Self-Scheduling)*, a new strategy that adapts the chosen allocation policy to a heterogeneous grid environment with local workload. Considering the heterogeneity and dynamic characteristics of the grid, *PSS* is able to modify the length of the work units during execution, based on average processing time needed to compare some database segments in each grid node.

The general expression used for work unit allocation is shown in (11). In this expression, $A(N,P)$ is the task allocation policy for a system with N workload units and P nodes and $\Phi(m,p_i,P)$ represents the weight calculated by *PSS*.

$$allocate(m,p_i,N,P) = A(N,P) * \Phi(m,p_i,P) \qquad (11)$$

To distribute database segments to nodes, the master analyzes periodic notifications sent by the slaves,

reporting the progress in processing work units. The expression used is $\Phi(m, p_i, P)$ (12), defined as the weighted mean from the last Ω notifications sent by each p_i slave node.

$$\Phi(m, p_i, P) = \frac{P * \left(\frac{\sum_{i=1}^{P} \Gamma(m, p_i, \Omega)}{\Gamma(m, p_i, \Omega)} \right)}{\sum_{i=1}^{P} \left(\frac{\sum_{i=1}^{P} \Gamma(m, p_i, \Omega)}{\Gamma(m, p_i, \Omega)} \right)} \quad (12)$$

The expression $\Gamma(m, p_i, \Omega)$ (equation 13) specifies the average computing time of a database segment in a node p_i, considering the last Ω notifications of $TE(m, p_i, \tau)$, which is the average computation time of τ work units (database segments) assigned by a master node m to a slave p_i, in units of work per units of time. The τ parameter indicates how many work units must be processed before a notification is sent to the master.

At the moment of the computation of Γ, if there is not enough notifications of TE, the calculation is done with total k notifications already received.

$$\Gamma(m, p_i, \Omega) = \frac{\sum_{j=1}^{min(\Omega, k)} TE(m, p_i, \tau)}{min(\Omega, k)} \quad (13)$$

5.3 PackageBLAST's General Architecture

PackageBLAST was designed as a grid service over Globus 3, based on Web Services and Java (figure 1).

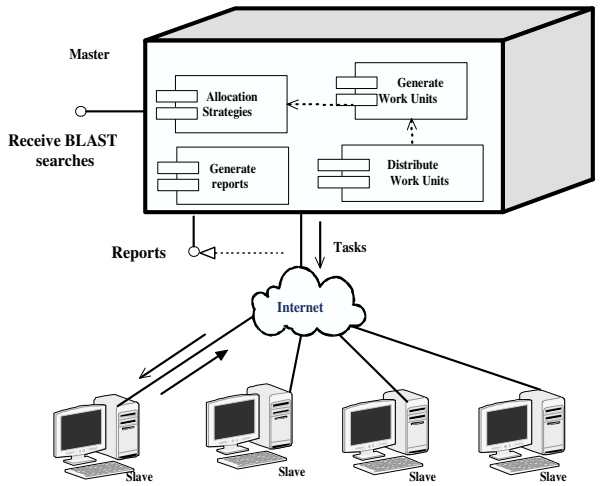

Figure 1. PackageBLAST architecture.

The module *Allocation Strategies* contains implementations for the pre-defined allocation policies - *Fixed, SS, GSS, TSS and FAC2* (section 3) - and also makes possible the creation of new allocation strategies.

The module *Generate Work Units* is the core of the PSS mechanism. It calculates the weight of each slave node and decides how many work units will be assigned to a particular slave node, according to the current allocation policy.

Distribute Work Units is the module that is responsible by the communication between the master and slaves nodes. Finally, the module *Generate Reports* obtains the intermediary outputs sent by the slave nodes through file transfer and merges them into a single BLAST output report.

In general, the following execution flow is executed. The master node starts execution and waits for slave connections. To start processing, a minimum number of slaves must register into the master node, by calling a master grid service. After receiving connections from the slaves, the master uses the Globus 3 notification mechanism to inform them about their initial segments to compare. The slaves process τ database segments and notify the master, which uses this information to compute the next allocation block size based on the selected allocation strategy and the weight provided by PSS. Then, the master sends a XML message to the slave informing its new segments to process. This flow continues until all segments are processed.

6 Experimental Results

PackageBLAST was evaluated in a 16-node heterogeneous grid testbed, composed by two laboratories (LABPOS and LAICO), interconnected by the University of Brasilia network at 100Mbps. Eleven desktops were used (P01-11) in LABPOS and four desktops (L01-04) and a notebook (NB) were used in LAICO. All grid nodes were Linux machines with Globus Toolkit 3.2.1, NCBI BLAST 2.2.10 and Java Virtual Machine 1.4.2. Hardware features of each group of machines are given in table 1.

Node Names	CPU	Memory	HD
NB	AMD64 3.2GHz	512 MB	80 GB
L01, L02, L03	AMD 1.7GHz	256 MB	30 GB
L04	PII 350MHz	160 MB	6 GB
P01, P02, P03, P04, P05, P06, P07, P08, P09, P10	AMD 1GHz	256 MB	20 GB
P11	AMD 900 MHz	128 MB	20 GB

Table 1. Characteristics of the grid testbed.

For the tests, we created heterogeneous grid configurations composed by 2, 4, 8 and 16 computing nodes. The machines that composed each grid configuration are listed in table 2.

In our tests, we used the genetic database *nr*, which has a size of 1.2GB, and was obtained from the NCBI

Grid configuration	Machines
2 nodes	NB, L04
4 nodes	NB, L01, P01, L04
8 nodes	NB, L01, L02, L03, P01, P02, P11, L04
16 nodes	NB, L01, L02, L03, P01, P02, P03, P04, P05, P06, P07, P08, P09, P10, P11, L04

Table 2. Grid Configurations

site. The query sequence used was the one called *takifugu rubripes alpha globin gene cluster*, obtained from the internet. This sequence is composed by nucleotides and has a size of 36KB. We also used a subset of this sequence in our tests, containing 10KB.

6.1 *PackageBLAST* Evaluation

In order to investigate the performance gains of *PackageBLAST*, we executed BLASTX in 2, 4, 8 and 16 grid nodes. Each BLAST search compared the 10KB DNA sequence against the *nr* genetic database segmented in 167 parts of 5 million characters each. Five allocation strategies were employed in the tests.

To calculate the absolute speedups, the BLAST sequential version was executed with the *nr* unsegmented database in machines with distinct hardware configurations. The total execution times obtained with the serial execution on each machine were used to compute speedups. Execution times for allocation strategies are presented in table 3.

Allocation Strategy	2 nodes	4 nodes	8 nodes	16 nodes
FIXED	2037	999	491	252
SS	1112	514	246	134
TSS	1296	570	259	143
GSS	1115	535	250	127
FAC2	1187	514	266	142

Table 3. Execution times (in seconds) for BLASTX execution in grid environment.

Figure 2 shows a plot of packageBLAST speedup, considering the best execution time for a given number of nodes. As can be seen in this figure, *PackageBLAST* achieved very good speedups. Considering the worst (L04), average (P01) and best (NB) node in the grid, the speedups obtained were superlinear, close to linear and sublinear, respectively. However, the speedup obtained in the NB case (11.28) for 16 machines, is a very good one, considering the differences in hardware among the machines.

In table 3, it can also be noticed that there is no allocation strategy that always reaches the best execution time. For instance, the best execution times for

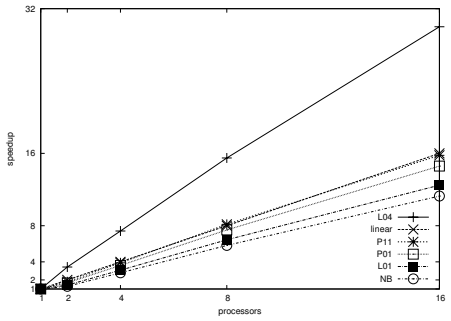

Figure 2. Absolute speedups for 2, 4, 8 and 16 nodes

8 and 16 nodes were obtained with the *SS* and *GSS* strategies, respectively.

To evaluate the *PSS* strategy, we executed *PackageBLAST* with the same BLAST search with 16 grid nodes, introducing local workload in nodes L01, L02, P01 and P02. The load was started simultaneously 30 seconds after the beginning of the BLAST search and consisted of the execution of the *formatdb* application over the *nr* database. Three scenarios were simulated: 1) with the *PSS* strategy, but without workload (PSS) ; 2) with the *PSS* strategy and workload (PSS 2x), to use grid environment knowledge obtained in the preceding iteration; and 3) Execution without *PSS* and with workload (without PSS). Data gathered in our experiments are presented in table 4.

Allocation Strategy	(1)with PSS	(2)PSS 2x	(3)without PSS	Gain
Fixed	316	184	393	113.59%
SS	186	177	179	1.13%
TSS	160	162	171	5.56%
GSS	149	159	339	113.21%
FAC2	156	165	153	-7.27%

Table 4. PSS strategy evaluation in 16 nodes with local workload.

As expected, the allocation strategies that assign a large amount of work to the nodes (*fixed* and *GSS*) obtained great benefit from using PSS (113.59% and 113.21%, respectively). This is due to the fact that a slow node can easily become a bottleneck in these strategies. For instance, when using the *fixed* strategy without PSS (table 4), 10 segments are assigned to each node. In our testbed, the slowest machine (L04) clearly became a bottleneck. In the second time PSS executes (table 4), the weight assigned to L04 was 0.59 and that made it process 5 segments instead of 10. Since *GSS* assigns at the beginning of the computation a great

amount of work to the nodes, a great benefit can also be obtained from PSS.

Among the remaining strategies, *TSS* was the one that obtained the best benefit from PSS (a reduction of 5.56% in its total execution time). *SS* obtained a very small benefit from using PSS (1.13%) and *FAC2* experimented an augmentation in its execution time when using PSS.

We repeated this test in a configuration with 8 machines, and the results obtained are shown in table 5. In this case, the load was introduced in nodes L01 and P01.

Allocation Strategy	(1)with PSS	(2)PSS 2x	(3)without PSS	Gain
Fixed	487	376	480	27.66%
SS	293	307	179	-4.56%
TSS	278	292	329	12.67%
GSS	275	289	331	14.54%
FAC2	281	288	298	3.47%

Table 5. PSS strategy evaluation in 8 nodes with local workload.

In table 5, the best performance gain was also obtained by the use of PSS with the *fixed* strategy. The use of PSS with *GSS* and *TSS* also obtained considerable gains (14.53% and 12.67%, respectively). As in the preceeding case, *FAC2* and *SS* obtained small benefit from PSS. However, the behavior observed for these two policies was the opposite of the previous case since, for 8 nodes, it was *SS* that experimented an augmentation on the execution time.

We varied the PSS parameters τ (number of database segments processed between two notifications) and Ω (number of notifications used to compute PSS) (section 3) to evaluate the PSS behavior in the following scenario in a four-node grid. In this experiment, a three-minute local workload (*formatdb*) was introduced in node P01 when the last task of the first *TSS* allocation starts. The goal here was to evaluate the impact of medium-lived local tasks in PSS. Figure 3 presents the gains obtained. In this figure, "PSS" represents the first execution of PSS, where the local load was introduced and "PSS 2x" represents the second execution of PSS. In scenario 1, "PSS 2x" does not contain local workload (short-lived task) while scenario 2 contains it (long-lived task).

In this scenario, the local load finishes its execution before the BLAST search ends. On the average, intermediate values for τ and Ω (2,2) led to better performance gains.

In *PackageBLAST*, the number of allocation messages exchanged by the master and a slave depends on the allocation policy and on the weight assigned to a

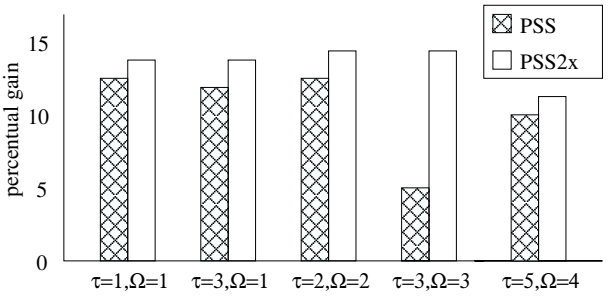

Figure 3. Percentual gain obtained by PSS varying τ and Ω parameters.

node. Table 6 presents the total number of allocation messages sent by the master node in the four-node grid (table 2) using *TSS* (section 3). The same variation of τ and Ω presented in figure 3 was analyzed here. The percentual gain in execution time obtained with PSS and with PSS2x are also presented.

Parameters	Gain PSS	Gain PSS2x	Msgs without PSS	Msgs PSS	Msgs PSS2x
$\tau = 1, \Omega = 1$	6.92%	10.21%	17	14	27
$\tau = 3, \Omega = 1$	6.10%	11.03%	17	13	22
$\tau = 2, \Omega = 2$	8.92%	9.54%	17	15	17
$\tau = 3, \Omega = 3$	3.63%	10.02%	17	13	18
$\tau = 5, \Omega = 4$	10.17%	10.38%	17	14	19

Table 6. Percentual gains in execution time and the total number of allocation messages

For all cases, in the first execution of PSS, the number of messages decreases, when compared to the execution without PSS (table 6). However, in the second execution of PSS, the number of messages is increased for all variations of parameters. Nevertheless, this augmentation leads to better execution times (Gain PSS2x). This indicates that PSS is assigning more segments of smaller size to the grid nodes. As can be seen in table 3, *SS* presents very good execution times in our environment so, despite the increase in the number of messages exchanged, PSS decides to reduce the size of the allocation blocks.

strategy	4-node 10KB	4-node 10KB	8-node 36KB	8-node 36KB
SS	514	2446	246	1319
TSS	547	2425	259	1328
GSS	515	2406	245	1329

Table 7. Execution time (seconds) obtained with 10KB and 36KB sequences

In order to evaluate the impact of the query sequence size on our mechanism, we used sequences of 10KB and 36KB in a four-node and eight-node grid with PSS and the SS, TSS and GSS strategies (table 7). In this test, we can see that the best execution times were obtained with GSS and SS, in our eight-node grid, with 10KB and 36KB query sequences, respectively. Also, we observed a great augmentation on the execution time. For a sequence 3.6 times larger, the average execution time augmented 5.3 times, in the eight-node grid. This occurs due to BLAST characteristics and a similar result has been reported in [6].

7 Conclusion

In this article, we evaluated *PackageBLAST*, an adaptive multi-policy grid service to execute master/slave BLAST searches in a grid environment.

The results collected by running *PackageBLAST* with five allocation policies in a 16-machine grid testbed were very good. In order to compare a 10KB real DNA sequence against the *nr* genetic database, we were able to reduce execution time from 30.88 min to 2.11 min. Also, we showed that, in our testbed, there is no allocation policy that always achieves the best performance and that makes evident the importance of providing multiple policies. Moreover, we showed that the introduction of PSS in a grid environment with local workload led to very good performance gains for some policies (mainly *fixed*, *GSS* and *TSS*).

As future work, we intend to run *PackageBLAST* in a geographically dispersed grid testbed in order to evaluate the impact of high network latencies in the allocation policies and in *PSS*. In addition to that, we intend to implement and evaluate the module *Generate Reports*, which is responsible to merge the results obtained by the slave nodes. We also plan to integrate a fault tolerance mechanism to *PackageBLAST*.

References

[1] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. A basic local alignment search tool. *Journal of Molecular Biology*, 215:403–410, 1990.

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

[3] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *International Journal of Supercomputer Applications*, 11(2):115–128, 1997.

[4] I. Foster and C. Kesselman. *The Grid: Blueprint of a Future Computing Infrastructure*. Morgan-Kauffman, 1999.

[5] S. F. Hummel, E. Schonberg, and L. E. Flynn. Factoring: A method for scheduling parallel loops. *Communications of the ACM*, 35(8):90–101, Aug. 1992.

[6] I. Korf, M. Yandell, and J. Bedell. *BLAST - An Essential Guide to the Basic Local Alignment Search Tool*. OReilly Associates, June 2003.

[7] A. Krishnan. Gridblast: High throughput blast on the grid. *Symposium on Biocomputing*, January 2003.

[8] J. Nabrzyski, J. Schopf, and J. Weglarz. *Grid Resource Management: State of the Art and Future Trends*. Kluwer Academic Publishers, 2003.

[9] D. Peng., W. Yan, and Z. Lei. Parallelization of blast++. Technical report, Singapore-MIT Alliance, 2004.

[10] M. K. Satish and R. R. Joshi. Gbtk: A toolkit for grid implementation of blast. *In proceedings of the High Performance Computing and Grid in Asia Pacific Region, Seventh International Conference on (HPCAsia'04)*, pages 378–382, January 2004.

[11] J. C. Setubal and J. Meidanis. *Introduction to Computational Molecular Biology*. Brooks/Cole Publishing Company, 1997.

[12] G. Shao. *Adaptive Scheduling of Master/Worker Applications on Distributed Computational Resources*. PhD thesis, University of California at San Diego, 2001.

[13] L. Smarr and C. L. Cattlet. Metacomputing. *Communications of the ACM*, 35(6):44–52, June 1992.

[14] T. Smith and M. Waterman. Identification of common molecular subsequences. *J. Mol. Biol.*, 147:195–197, 1981.

[15] P. Tang and P. C. Yew. Processor self-scheduling for multiple nested parallel loops. In *Int. Conf. on Parallel Processing (ICPP)*, pages 528–535, 1986.

[16] T. H. Tzen and L. M. Ni. Trapezoidal self-scheduling: A practical scheme for parallel compilers. *IEEE Transactions on Parallel and Distributed Systems*, 4(1):87–98, Jan. 1993.