Two-Level Parallelism to Accelerate Multiple Genome Comparisons

Oscar Torreno(✉) and Oswaldo Trelles

Department of Computer Architecture, University of Málaga,
Campus de Teatinos, 29071 Málaga, Spain
{oscart,ortrelles}@uma.es

Abstract. We present a two-level parallel strategy focused in the enhancement of GECKO software for multiple and pairwise genome comparisons. GECKO was developed to break the computational barriers on search space and memory demands faced by equivalent software. However, although being faster than equivalent software for comparing long sequences, its execution time attracted our interest to develop a parallel strategy. Additionally, the execution time is even higher in multiple genome comparisons where several independent pairwise comparisons are typically performed sequentially. After a careful study of the internal data dependencies of the GECKO modules, we noticed that most of them were subject to an easy and efficient parallelization. The result is a two-level parallel approach to accelerate multiple genome comparisons. The first level is aimed at parallelizing each independent pairwise genome comparison of a multiple comparison study to a different core. This level is application-independent, we are using GECKO but any other equivalent software can be used. The second level consists on the internal parallelization of GECKO modules with evident enhancements in performance while results remain invariant. After solving the problems of combining the big amount of I/O operations overlapped with computation, the obtained speedups reflect the good efficiency of the devised strategy.

1 Introduction

Two-level and more generally multi-level parallelism have been already applied in a number of different fields including video coding, aerodynamics and shape design [3,6,7]. After analysing the specifics of the application to be parallelized, these multi-level approaches use either only message passing implementations applied to the different levels, or merge the usage of MPI with OpenMP for multi-core shared memory machines. In addition, some approaches such as [7] use hybrid CPU-GPU implementations to accelerate the computation. Examples of reasons motivating the use of these multi-level strategies are the mixture of lightweight and heavyweight tasks or the dissimilar combination of computation and I/O operations.

We have found that multiple and pairwise genome comparisons have the mentioned computation patterns and are therefore suitable to be parallelized

DOI: 10.1007/978-3-319-58943-5_36

in multiple levels. The computational space and memory demands of current software is significantly high for just comparing two long sequences. GECKO [9] was designed to overcome these limitations of equivalent software. However, although reporting shorter execution time compared to equivalent software, its execution time is still big enough to study potential parallel approaches.

The problem becomes even more notorious when we aim at comparing a set of genome sequences in what is referred as a multiple genome comparison study. For such studies, pairwise genome comparison software is used, executing it sequentially several times depending on the number of genomes under study. If already at pairwise genome comparison level the execution time is important, at the multiple genome comparison level it becomes even higher and therefore more interesting from the computational point of view.

After analysing in more detail the parallelization possibilities, we noticed it was possible to apply a two-level strategy. First, we observed that each pairwise genome comparison inside a multiple genome comparison is independent so it represents an embarrassingly parallel problem. Second, and less obvious, we detected that some of the GECKO modules were subject to be parallelized. For instance, the first step of GECKO calculates a dictionary of K-mers (words of length K), which may consume a significant amount of time for long sequences. In this case, splitting the calculation of given word prefixes to different processors could be a solution. Similarly, the rest of GECKO steps could be parallelized with relatively simple strategies.

In this work, we present a two-level parallelization strategy designed to speed up multiple and pairwise genome comparisons. Efficiency in both levels has been obtained by using dynamic scheduling algorithms which generate the sufficient number of tasks to overlap I/O and computation. Hybrid solutions using GPUs are not considered in this work (but will be possibly considered in the future) because the application is data intensive and the loading time to GPU memory would govern the performance. The resulting strategy reduces the execution time when the number of processors increase, especially while working with long sequences. The code has been developed in C using the Open MPI library [1] (it will be available under request). The associated binaries can be obtained from http://bitlab-es.com/gecko.

2 Related Work

In the literature, several papers provide a review on HPC solutions applied to pairwise and multiple sequence comparisons such as [8]. The solutions mentioned in these papers use different architectures such as GPUs, FPGAs, multicores and/or Intel Xeon Phi being able to compare at most sequences of up to hundreds of millions of characters (10^8), as stated in [8]. The CPU implementations employ fine-grained parallelism using different data distribution techniques. Some of the CPU techniques report memory consumption of quadratic order, thus limiting the size of the input sequence. FPGA implementations clearly accelerate the process but they have important limitations. First, they can only handle sequences up to 10^5 base pairs (bp), and second, almost all of them only report the alignment score as output. GPU-based solutions accelerate the process as well, but they also are limited in the length of the input sequence (10^8 bp). In addition, most GPU implementations require quadratic space to report the alignment. A small number of Intel Xeon Phi implementations exist, which already report better performance than GPU implementation in some cases.

Trying to benefit from the best part of each architecture, there already exist a number of hybrid approaches using various devices simultaneously. Such solutions implement two-level parallelization, in which in a first coarse-grained level a set of sequences is assigned to each device. The second level (i.e. fine-grained) often uses previously proposed algorithms/tools to compare the sequences. We considered this two-level approach suitable to multiple genome comparisons using GECKO. The reason why we are not using GPUs, FPGAs or Intel Xeon Phi in this work is because of the mentioned input sequence size limitation faced in such devices.

As reported in [8], even with the performance improvement of the HPC techniques applied to the research field, the comparison of long sequences such as human chromosomes still takes more than 9 h to complete. Therefore, our effort in this paper concentrates in both reducing such execution time, removing the input sequence size limitation and reporting the alignment together with quality information such as the score (i.e. the main limitations of related approaches).

3 System and Methods

3.1 The Pairwise Genome Comparison Application

The latest release of GECKO program has been used in this work as the base sequences comparison algorithm (see Algorithm 1). GECKO is a modular application which calculates a set of conserved segments or High-scoring Segment Pairs (HSPs) shared by two given input sequences. It has not been modified in terms of functionality, however a number of minor changes were introduced in order to be able to parallelize it.

The program starts by calculating a dictionary of words of length K (K-mers) for each of the input sequences to be compared. The dictionary calculation scans the sequence with a sliding window of length K and a step of 1 producing an alphabetically sorted dictionary of K-mers together with their frequencies and occurrence positions.

Using the dictionaries of both input sequences, a set of exact word matches (seed points or hits) is produced. A hit is defined by the coordinates of the same word in the input sequences, therefore, a given word W_i with frequencies f_1 and f_2 in sequences 1 and 2 respectively, will produce $f_1 \times f_2$ hits following all the combinations.

In order to reduce the previously generated hits set, hits are sorted and optionally filtered based on proximity. The next program compares the residues present at each sequence starting where each hit occurs, adding or subtracting a given value depending if the residues match or not. When the alignment score becomes negative, the calculated alignment (using the maximum score reached) is reported in case it passes the threshold parameters.

Algorithm 1. GECKO

```
1: Calculate a dictionary of K-mers for each input sequence.
2: Calculate a set of seed points based on the previously calculated dictionaries.
3: Sort (and optionally filter) the produced set of seed points.
4: Calculate the final set of HSPs based on the previous set of seed points.
```

As described in GECKO documentation, CPU time is mostly concentrated in the sorting procedures ((1) and (3)) due to the amount of data to be sorted what forces the program to use the hard disk. Therefore, our main effort will be in the parallelization of such steps. Besides, steps (2) and (4) are also subject to parallelization although they do not concentrate the major CPU time. The parallelization speedup for these steps will be noticed mostly for long similar sequences. All the steps (from (1) to (4)) have to be executed sequentially since each step is using the output of the previous. However, there is still room to internally parallelize each step.

3.2 Generic Overview of the Parallelization Approaches

Master-Slave. This approach has been applied to the two parallelization levels. In the first level, we have applied a master-slave tasks distribution approach to perform each pairwise genome comparison of a multiple genome comparison study in a different core (see Fig. 1.A). In the second level, the slaves calculate the partial result of the modules composing GECKO (see Fig. 1.B). This approach considers at both levels as many slave processes as cores being used. The master process reads the set of tasks from a workload file, which is generated by a previous mapping process. Later the master distributes the tasks, assigning the cores more tasks as soon as they become idle.

Considering the high number of I/O operations performed by GECKO modules, the master is assigning more than one task per core in order to overlap I/O and computation. Additionally, this is done to reduce the overhead introduced by sending tasks in separated messages. Depending on the number of processes and the selected prefix size (as explained in Sect. 3.3), the *task_per_core* value is either 2 (for number of cores power of 2) or 4 (for number of cores power of 4) in order to always have more than 1 task per core.

Balanced Splits Distribution. This strategy is similar to the master-slave approach, but in this case the master and slaves are customised for the specific parallelized module, what contrasts with the generic ones of the previously described strategy. Besides, the master calculates the offset coordinates of a balanced set of independent data chunks, which are later processed by the slaves.

Once all the partial results become available, the master produces the final output. As in the previous approach, each processor is assigned either with 2 or 4 data chunks depending on the number of cores.

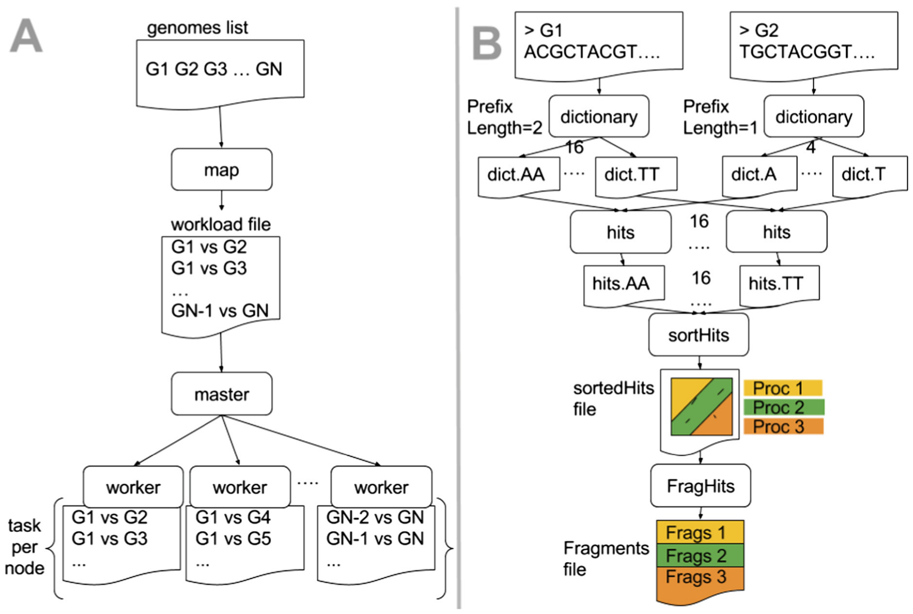

Fig. 1. Overview of the parallelization levels. Sub-figure A outlines how the strategy starting from a list of genomes ends performing each independent pairwise genome comparison in a separate worker. Sub-figure B shows the performed parallelization within the internal modules of GECKO.

3.3 Details of the Parallelization Strategies of the Second Level

Parallelization Strategy for the 'Dictionary Step' [Step (1)]. The parallelization of this step is performed in two levels. The first and simpler level, is the parallelization at sequence level, since the dictionary calculation of each sequence is independent. The second level, splits the dictionary calculation in N tasks, being $N = 4^{PrefixSize}$. $PrefixSize$ indicates the size of the prefix to be used to split the work, so special care with its value must be taken in order to have the correct number of tasks as explained in the Sect. 3.2. The alphabet used for the K-mers is $\Sigma = \{A, C, G, T\}$, so when the parallelization is made with $PrefixSize = 2$, 16 tasks are generated. Such tasks calculate words starting with the following prefixes (in alphabetical order): $AA, AC, AG, AT, ..., CA, CC, ..., TT$.

Parallelization Strategy for the 'Hits Step' [Step (2)]. This step calculates the matches between the N previously calculated sub-dictionaries suitable to produce matches. For example, if the dictionaries were calculated with $N = 16$ then we have 16 comparisons, reference.dict-AA against query.dict-AA, reference.dict-AC against query.dict-AC, etc. The workload generation when the dictionaries were calculated with the same parameters in both cases is straightforward, but when they were calculated with different values then it is a little bit more difficult. For instance, if the dictionary was calculated using $N = 4$ and $N = 16$ respectively for each of the input sequences, the tasks are: reference.dict-A against query.dict-(AA, AC, AG, AT), reference.dict-C against query.dict-(CA, CC, CG, CT), etc. At the end of the computation, a reduce task just concatenates each partial output file. This last step needs to be done because later the sorting step requires the input data in one single file.

Parallelization Strategy for the 'Sorting Hits Step' [Step (3)]. This step has been parallelized with a message-passing implementation of the quicksort algorithm. The master sends to the workers the coordinates of the file that they have to sort. The workers sort these parts and write their partial sorted chunk. Once all the chunks are sorted, the master assigns merging tasks to the workers following a hierarchical merge algorithm.

Parallelization Strategy for the 'FragHits Step' [Step (4)]. In this case the parallelization strategy splits the input hits file in groups of diagonals (i.e. an arbitrary value defined as the difference of the positions in the query and reference sequences respectively). The reason behind this data splitting strategy is that hits belonging to the same diagonal have data dependencies. The extension of a hit could cover a further one within the same diagonal and this covered hit should in turn not be extended because it will produce a fragment contained in the previous one. In order to have a balanced set of tasks the number of diagonals varies depending on the numer of hits they contain. A final reduce step concatenates the partial results producing a unique HSPs file equivalent to the one generated by the sequential version.

4 Results

In this section, the performance of the applied paralellization strategies in terms of speedup and reduced time is illustrated. All the speedup curves contained in this document reflect the average execution time of 10 runs. A number of different tests are used to illustrate the performance achievement on each of the parallelized levels. These tests are using input data ranging from short to large sequences and also different *tasks_per_core* values due to the high number of I/O operations performed by some modules. In addition to the performed tests to each step of the second parallelization level, the multiple genome comparison and overall application speedups are shown to illustrate the gains achieved by the presented two-level parallelization strategy.

4.1 Infrastructure

This new implementation has been tested in the fat nodes of the Picasso supercomputer located at the University of Málaga (Málaga, Spain). Each fat node has 2 TB of RAM and eight Intel E7-4870 processors, which deliver 96 Gflop/s. For the first parallelization level only one node has been used until the measurement of 64 cores and two nodes for the 128 cores measurement. For the second parallelization level only one node has been used, requiring no MPI communication over the network, since each node has 80 cores and the speedup measurements are made until 32 cores.

4.2 Dataset

The selected test datasets contain several public available sequences[1] of different sizes in order to thoroughly study the speedup of the two parallelization levels. In the first level, we are using two sets of 30 and 40 small sequences of Mycoplasma genus with an average length of 1 Mbp. Both sets contain sequences sharing different level of similarity ranging from closely to remotely related sequences. The dataset to test the internal parallelization level is composed of bacteria and mammalian sequences ranging from 5 to 410 Mbp. The large mammalian sequences (Homo sapiens (HS) and Macaca Mulatta (MM)) are used in two tests. The first test uses the chromosome 1 of both mammalian sequences (from now on the test will be referred as HS-MM(chr1)), while the second one compares the concatenation of chromosomes 1 and 2 of each species in order to conform a longer sequence (the test will be referred as HS-MM(chr1+2)).

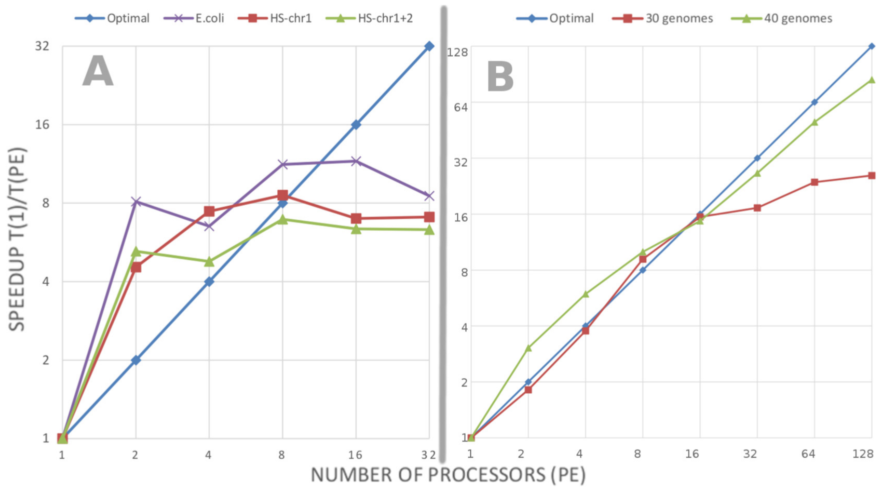

Fig. 2. A: Total application speedup; B: Multiple genome comparison speedup

[1] http://www.ncbi.nlm.nih.gov/genome/.

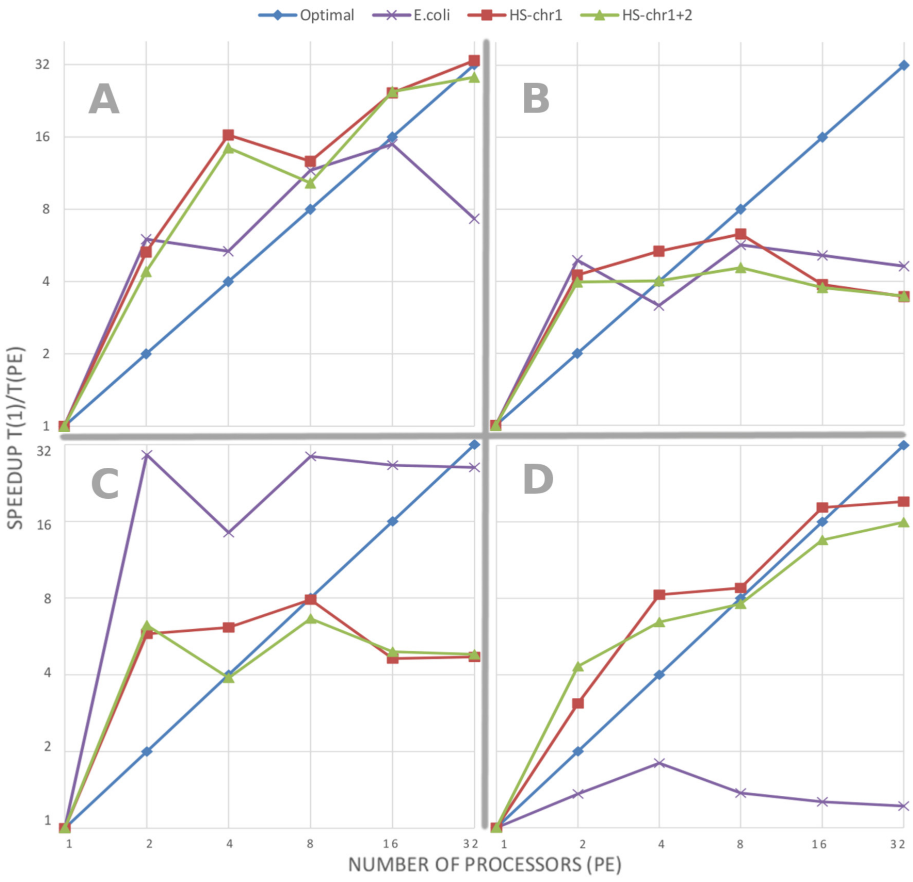

Fig. 3. A: Dictionary step speedup; B: Hits step speedup; C: Sort hits step speedup; D: FragHits step speedup.

4.3 Speedup of the Performed Two-Level Parallelization

The parallelization strategy of the first level applied to enhance multiple genome comparisons follows the speedup curve shown in Fig. 2.B. This figure contains two series, which show the speedup of the all against all comparison of the 30 and 40 genomes sets described in Sect. 4.2. The speedup curve of the 30 genomes set accounts for 435 pairwise genome comparisons, whereas the second curve of 40 genomes set comprises 780 pairwise genome comparisons. It is worth mentioning that an all vs. all comparison of N genomes accounts for $N*(N-1)/2$ comparisons given the symmetry property of a pairwise comparison.

The devised strategies in the second parallelization level follow the speedup curves shown in Figs. 3 (A, B, C and D; which relates in order to the steps of Algorithm 1) and 2.A, which shows the overall application speedup.

5 Discussion

5.1 Speedup of the Multiple Genome Comparison Study

The speedup of the first parallelization level (see Fig. 2.B) indicates that the application is scalable. It is worth noting that until 16 PE, both series (i.e. 30 and 40 genomes) have a speedup close to the theoretical one or even super-linear because of the overlap between I/O and computation produced while executing several comparisons at the same time. From 32 PE onwards, the speedup of the 30 genomes series degrades because there are not enough tasks for the available cores. In contrast, the 40 genomes series, which generates a higher number of tasks, keeps scaling closer to the theoretical speedup with an efficiency of 65.98% in the worst case.

5.2 Dictionary Step Speedup

Results are good when the sequence size is big enough as can be observed in Fig. 3.A, having reached accelerations above the theoretical until 8 PE for E.coli, 32 PE for HS-MM(chr1) and 16 PE for HS-MM(chr1+2). The performance reduction that can be observed in the E.coli series is produced by its short length, which is on of the parameters that conducts the compute time. It is worth noting that the speedup of the longest sequence (HS-MM(chr1+2)) is not the best as it is supposed to be, based on previous assumptions, mainly because of the higher I/O load. It is also important to note that the application scales good until 16 PE, from where the super-linear speedup turns into normal speedup. We believe that the cause of this behaviour could be that the required number of I/O operations is very high, what is not giving a good computation-I/O ratio. Furthermore, since processes within the same physical node share the filesystem, this could be also a bottleneck in this case.

5.3 Hits Step Speedup

We can observe two interesting aspects in the speedup curves of the hits step (see Fig. 3.B). First, the super-linear speedup achieved in the cases of 2 and 4 PE for the longer sequences (HS-MM(chr1+2) and HS-MM(chr1)). This is caused again by the fact of having simultaneous executions in each node overlapping computation and I/O operations. The second aspect, the reduction of the speedup just after obtaining the super-linear one (after 4 PE in one case and 2 PE in the other cases). Here, we believe that writing the output file is consuming most of the time due to its size (around 78 GB for HS-MM(chr1) and 252 GB for HS-MM(chr1+2)). In consequence, adding more cores, which provide processing power, does not speed up the process. With regards to the shortest sequence (E.coli), the output file is much smaller, but the situation remains the same, because in essence the ratio between compute and I/O is similar.

5.4 Sort Hits Step Speedup

Similarly to the hits step, we can also observe in the speedup of this module (see Fig. 3.C) a super-linear speedup until 16 PE for E.coli, 4 PE for HS-MM(chr1) and 2 PE for HS-MM(chr1+2). However, in this case the reason resides in the data fitting in main memory instead of working with it stored on the hard disk. Although the curve shapes are similar, the speedup achieved in the case of HS-MM(chr1) for 4 PE (6.13) is higher than the obtained in the hits step (5.34). Besides super-linearity, the speedup is again reduced due to the size of the files to be read and written. In constrast, in the short sequences series, the super-linear speedup is also achieved for 16 PE, what confirms our assumption of the bottleneck caused by the size of the output file.

5.5 FragHits Step Speedup

For this step, in the case of long sequences, the speedup is again super-linear until 4 PE for the HS-MM(chr1+2) case and until 16 PE for HS-MM(chr1). Although the speedup is super-linear at the beginning, the efficiency level degrades, resulting in 60.06% in HS-MM(chr1) and 49.84% in HS-MM(chr1+2) with 32 PE. Again the results for short sequences demonstrate that the computational workload is not big enough in such case to take profit of a parallel strategy.

5.6 Overall Application Speedup in a Pairwise Comparison

The speedup curves shown in Fig. 2.A confirm what we explained in previous sections. The application reports super-linear speedup until 8 PE (except for HS-MM(chr1+2), reasons in Sects. 5.3 and 5.4). In addition, starting from 16 PE, the performance does not improve, what is normal due to parallelization overheads compared to the actual computation as well as the high I/O load the application has. It is important to note that although for many steps the speedup for the shortest sequence was not that good, the efficiency of the overall application is acceptable until 16 PE (61.25%) and the speedup is even better than the one of the two long sequences, mostly because the speedup is better in the most time consuming step (i.e. sort hits).

5.7 Time Reduction

Although the speedup curves in some cases suggest not particularly good efficiency levels, the time reduction has been considerable. In the first parallelization level, although the efficiency for the 30 genomes series is not good, the execution time has been reduced to 10 s compared to the 250 s of the sequential execution. In the second parallelization level we can observe a similar situation. Although for some GECKO modules the speedup is not good, the overall time reduction is significant. For instance, for the case of HS-MM(chr1+2) using 16 PE the time has been reduced to 49 min from the 5 h and 11 min of the sequential execution.

6 Conclusions

In this work we have approached the parallelization of multiple genome comparisons following a two-level strategy. The first level is aimed at parallelizing each independent pairwise genome comparison of the multiple comparison study to a different core. The second level consists on the parallelization of GECKO modules with evident time reduction while results remain invariant. Although the second parallelization level is customised for GECKO, the first one is generic enough to be used with any of the GECKO equivalent applications. The reason behind selecting GECKO is that it produces results of higher quality without computational barriers compared to current top methods such as MUMmer [5] or Mauve [2] as stated in [9].

To decrease the scheduling cost in the master process we implement a simple mapping of tasks to the available workers, assigning them a new set of tasks as soon as they became idle. This scheduler introduces a tasks per core parameter which allows the user to overlap the execution of tasks, what we found specially useful in terms of performance while working with a disparate combination of CPU and I/O bounded applications. In fact, the overlapping of I/O and computation is producing the super-linear speedups shown in the figures included in this document.

Tests in the first parallelization level have been performed using two different datasets of 30 and 40 genomes respectively. The selected dataset in this case represents the most typical multiple genome comparison study, which is the comparison of short sequences given that around 75% of the available sequences are short sequences[2]. However, the obtained results indicate that it would be possible to use long sequences as well. The obtained speedup in the 30 genome sequences set indicates that from 32 PE onwards the number of tasks is not sufficient. However, in the 40 genomes set the speedup maintains good efficiency levels beyond that point.

In the second parallelization level, tests using different sequence lengths have been performed, since this is one of the parameters governing the execution time. The obtained results show that all GECKO modules reduce significantly their execution time, although in terms of speedup with a high number of cores the results are not specially prominent. Analysing the speedup, we can extract the correct number of PE for each of the modules. In the case of the dictionary module this value is 32 PE (although for the short sequence the efficiency is not good). The hits and sort hits steps report asymptotic speedups, which are good in terms of efficiency until 8 PE in both cases. For the last application module (i.e. FragHits), the efficiency is acceptable until 32 PE for the long sequences cases and clearly not worthing to be parallelized for the short sequences case. The different values of number of PE suggest that the use of auto-scaling architectures such as cloud computing could be suitable for this application.

It is worth noting that the biological problem addressed here is really important. In fact, in comparative genomics the core applications include the

[2] https://gold.jgi.doe.gov/statistics.

competitors of GECKO (e.g. MUMmer, Mauve, Lastz [4]). Using this two-level parallel strategy for multiple genome comparisons, the researchers have a faster way to study a given sequence. The original version of GECKO was already able to compare two concatenated chromosomes in less time than parallel methods, which take 9 h as reported in [8]. The parallel version presented in this document reduces the execution time even further reducing it to 49 min. Additionally, this faster way of comparing multiple genomes allow users contrasting the current evolutionary models.

As future work, we plan to test it with more input sequences and in a different system in terms of number of processors and underlying filesystem. We sincerely hope, that this tests will reinforce the fact of the results obtained with the devised parallelization strategies described in this document.

Acknowledgements. This work has been partially supported by the European projects Mr. Symbiomath (grant no. 324554) and Elixir-Excelerate (grant no. 676559), and the Spanish national projects "Plataforma de Recursos Biomoleculares y Bioinformáticos" (ISCIII-PT13.0001.0012) and RIRAAF (ISCIII-RD12/0013/0006).

References

1. Open MPI. https://www.open-mpi.org/
2. Darling, A.E., Mau, B., Perna, N.T.: progressivemauve: multiple genome alignment with gene gain, loss and rearrangement. PLoS One **5**(6), e11147 (2010)
3. Duvigneau, R., Kloczko, T., Praveen, C.: A three-level parallelization strategy for robust design in aerodynamics. In: Proceedings of 20th International Conference on Parallel Computational Fluid Dynamics, pp. 379–384 (2008)
4. Harris, R.: Improved pairwise alignment of genomic DNA. Ph.D. dissertation, The Pennsylvania State University (2007)
5. Kurtz, S., Phillippy, A., Delcher, A.L., Smoot, M., Shumway, M., Antonescu, C., Salzberg, S.L.: Versatile and open software for comparing large genomes. Genome Biol. **5**(2), R12 (2004)
6. Marco, N., Lanteri, S.: A two-level parallelization strategy for genetic algorithms applied to optimum shape design. Parallel Comput. **26**(4), 377–397 (2000)
7. Momcilovic, S., Roma, N., Sousa, L.: Multi-level parallelization of advanced video coding on hybrid CPU+GPU platforms. In: Caragiannis, I., et al. (eds.) Euro-Par 2012. LNCS, vol. 7640, pp. 165–174. Springer, Heidelberg (2013). doi:10.1007/978-3-642-36949-0_19
8. Sandes, E., Boukerche, A., Melo, A.: Parallel optimal pairwise biological sequence comparison: algorithms, platforms, and classification. ACM Comput. Surv. (CSUR) **48**(4), 63 (2016)
9. Torreno, O., Trelles, O.: Breaking the computational barriers of pairwise genome comparison. BMC Bioinf. **16**(1), 1 (2015)