P-Aevol: an OpenMP Parallelization of a Biological Evolution Simulator, Through Decomposition in Multiple Loops

Laurent Turpin, Jonathan Rouzaud-Cornabas, Thierry Gautier, Christian Pérez

P-Aevol: an OpenMP Parallelization of a Biological Evolution Simulator, Through Decomposition in Multiple Loops

Laurent Turpin[1,2], Thierry Gautier[1], Jonathan Rouzaud Cornabas[2], and Christian Perez[1]

[1] Univ. Lyon, Inria, CNRS, EnsL, UCBL, LIP, France
laurent.turpin@inria.fr thierry.gautier@inrialpes.fr
christian.perez@inria.fr
[2] Univ. Lyon, INSA Lyon, Inria, CNRS, UCBL, LIRIS, France
jonathan.rouzaud-cornabas@inria.fr

Abstract. This paper presents how we have achieved the parallelization of Aevol, a biological evolution simulator, on multi-core architecture using the OpenMP standard. While it looks like a simple for-loop problem with independent iterations, the stochastic nature of Aevol makes the duration of the iterations unpredictable and it conveys a high irregularity. Classical scheduling algorithms of OpenMP runtimes turn out to be inefficient. By analysing the origin of this irregularity, this paper present how to transform the highly irregular Aevol for-loop to a sequence composed by a small duration irregular for-loop followed by work intensive for-loop easy to schedule using classical LPT algorithm. This method leads to a gain up to 27% from the best OpenMP loop schedule.

Keywords: Loop scheduling · Irregular iterations · Multi-core · OpenMP · *in-silico* simulation.

1 Introduction

Scientific applications made the development of High Performance Computing more and more relevant. Frequently, these applications are based on independent iterations loops. Aevol is an example of such application. The purpose of Aevol is to simulate millions of generations of an evolving population of micro-organisms. Each generation consists of a for loop iterating over the population. For each individual, the model simulates their evolution through stochastic selection and mutations that consist on random modifications of their structures. Our goal is to parallelize with OpenMP the evolutionary loop of Aevol, that is, at first glance, a simple for-loop with independent iterations.

The OpenMP API standard proposes 3 loop schedulers: static, dynamic and guided. Due to the Aevol stochastic model, the irregularity of the application requires a dynamic scheduler, like other scientific applications [18,1], in order to well balance the workload between the threads of the parallel region.

We, as most of HPC developers, are concerned with the following questions. Can better schedule be computed for our application? What would be the gains? How to implement it? The underlying problem was at first glance a list scheduling problem with unknown duration of tasks. Thanks to an analysis of the application structured as compositions of functions, we refine the problem by decomposing the loop in two sub-loops: the first one being a loop scheduling problem with unknown durations that permits to estimate the iteration's duration of the second loop. By doing so, it becomes possible to use a clairvoyant list scheduling algorithm. Using a method inspired by LPT scheduling [11] for the second loop, and thanks to a well balanced first loop and a limited impact of sorting tasks, we gain up to 27% more performance than the OpenMP dynamic.

The remainder of this paper is organized as follows: Section 2 introduces the Aevol software, how its computation is structured and how it is characterized. Section 3 explains the methodology developed to split the loop and how we use the application data to design a new scheduling method and some practical implementations done with OpenMP. Section 4 deals with the experimental evaluations of these implementations. Related work is discussed in Section 5. Last, Section 6 concludes the paper and opens up some future work.

2 Aevol, an Irregular Stochastic Program

This section presents Aevol, a computational biology software. It describes its computational model and characterizes the underlying main loop to parallelize.

2.1 Aevol: A Simulation of Darwinian Evolution

Biologists run *in-vitro* experiments in Petri dishes to observe the growth and evolution of simple organisms [3]. Aevol[3] proposes to run the same kind of experiments but *in-silico* [13]. Aevol is a C++ implementation of the biological model presented in [15]. It makes use of a stochastic model that puts uni-cellular organisms in a well defined environment and lets them evolve. Each organism, or individual, encloses genetic code in the form of a sequence of characters representing its DNA. The information contained in the genetic code is treated and eventually forms the phenotype of the individual, *i.e.*, its macroscopic behaviour or appearance. The phenotype is compared to an environmental target that models the environment where the micro-organisms leave. The difference tells how well the individual fits the environment. In Aevol, the *fitness* is represented as a scalar. For each generation, the whole population follows the three evolution steps:

1. **Selection:** the fitness of each individual is computed. The higher the fitness of an individual, the higher its probability of reproduction.
2. **Reproduction:** the survivors have the opportunity to pass their genetic code to their offspring. It is the principle of heredity.

[3] http://www.aevol.fr/

3. Mutation: the new-born individuals may get random variations on the genetic code that may modify their phenotype and their fitness.

The world in which individuals evolve is a 2D toric grid. In each cell of the grid lies one and only one individual. For each cell, competition is made among the 9 individuals of its neighborhood and only one of these 9 individuals is selected as the reproducer for this cell. That means that a single individual can reproduce multiple times within its neighborhood. After selecting all the reproducers for the entire grid, the population is wiped out and the reproducers are copied in the cells where they reproduce. Then mutation may occur randomly, depending on a mutation rate parameter defined by the user, on the DNA of the new population. Their new phenotype and fitness are computed.

At runtime, the computation time of a generation is very short *i.e.*, around 10 ms, thanks to a simple model and a small population. For instance in [16], the population size was 1024 individuals for a 32×32 square grid with different mutation rates (10^{-4}, 10^{-5} and 10^{-6}). But they computed for a total of 81 millions of generations demanding weeks of computations. Improving the performance of one generation becomes essential especially since Aevol is evolving toward a more complex model [17].

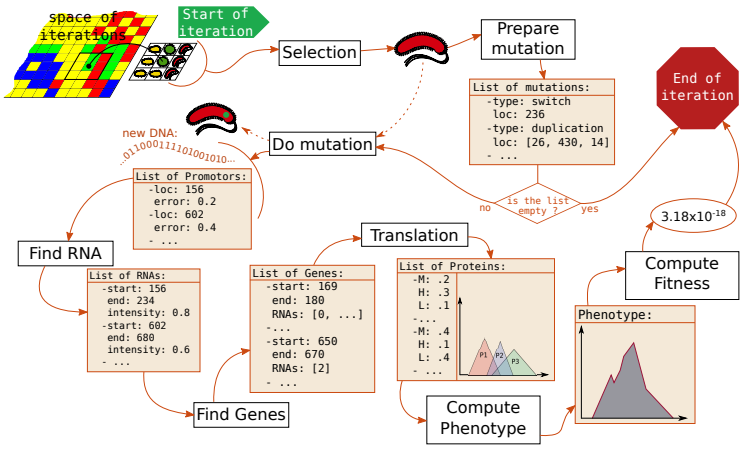

Fig. 1. Workflow followed by each cell for one generation. The data (orange boxes) pass through the different functions (white boxes). The output is the *fitness* value.

2.2 Computational Workflow of Aevol

As mentioned previously, Aevol runs forward generation by generation. For each generation n, the population of generation $n-1$ is known. The evolutionary loop iterates on each cell. The *selection* step is a *stencil* computation. Each cell clones the selected individual (DNA, phenotype and *fitness*) from neighborhood cells

from generation $n-1$. In that way, all cell computations are independent and can be computed in any order. At the end of the generation, individuals have to wait for their neighbors to start the new generation. However, a global barrier is used to simplify the synchronization at the end of each generation.

Once the selection is done, the remaining computation of an iteration can be viewed as a sequential workflow, as shown in Figure 1. An individual has a probability to mutate depending on the size of its DNA and the mutation rate specified as an user parameter. There are several kinds of mutation such as a one bit modification of the genetic code or duplication of the entire genetic code or even more. If an individual does not change, then all its information is known by its parent. For the others, so called mutants, their new DNA must be processed to compute their new fitness. It consists mainly on reading and recognizing sequences of characters. These functions are mostly memory-intensive computation.

2.3 Dynamic Characterization of the Computation

The time to process an individual (*i.e.*, a single iteration of a generation) strongly depends on its data (DNA) and if it is a mutant or not. The computation loop over the individuals is said to be *irregular* and it requires dedicated scheduling as discussed hereafter.

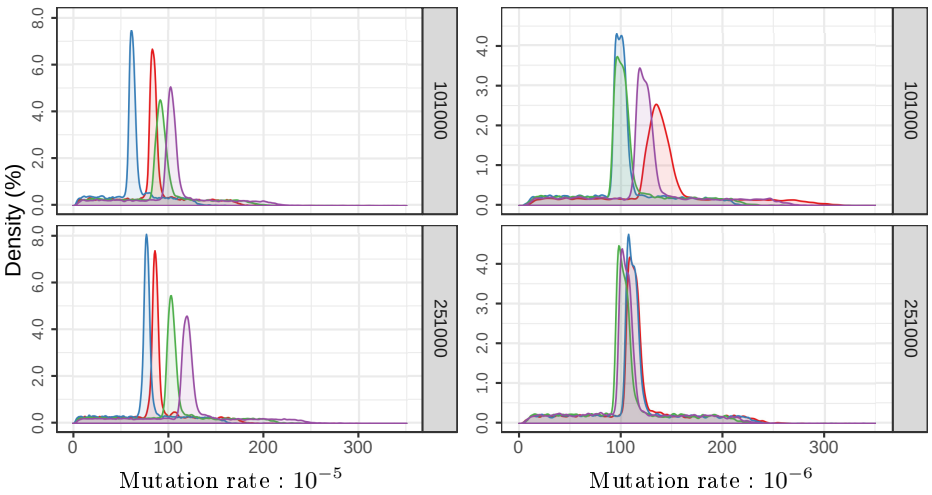

Fig. 2. Density in function of the time to process mutants (in μs). The colors correspond to distinct experiments. Top (bottom) plots are extracted at generation 101,000 (resp 251,000). Time scale is cut at 350 μs as very few (0.01%) individuals last longer.

The first origin of irregularity comes from the distinction between mutants and non-mutants which do not follow the same process. Non-mutants are significantly faster to compute *i.e.*, around 1% of the total runtime of a generation.

This proportion depends on the mutation rate which influences the number of mutants. Nevertheless, mutants always count for the large majority (99%) of the computation time. Besides, even among the mutants, we observe a large irregularity. As shown in Figure 2, a high density of individuals takes a similar amount of time to be processed, but there are still some individuals that can last up to 10 times longer than the others. In addition the distribution of these iterations varies depending on the generation and experimental parameters specified by the user. It is especially true for the mutation rate that can be strongly linked to size of the DNA [8]. Last, because the evolutionary model is stochastic, it is not possible to know which and how individual will mutate before its computation. The duration of each individual is therefore *a priori* unpredictable.

3 Parallelization of the Evolutionary Loop

To accelerate the simulation time, it is necessary to parallelize the evolutionary loop. The main issue is a loop scheduling issue, with time per iteration at fine grain. We limit our presentation to multi-core architecture for which OpenMP is an acceptable parallel environment with good performance.

```
1 /* original pattern in
      Aevol  */
2 for i = 1..N do
3   fitness[i] = compute(indiv[i])
```

Listing 1.1. Evolutionary loop.

```
1 #pragma omp parallel loop \
2   schedule(<arguments>)
3 for i = 1..N do
4   fitness[i] = compute(indiv[i])
```

Listing 1.2. First parallelisation.

3.1 Straightforward performance with OpenMP loop schedulers

With OpenMP, a direct parallelization is to add a `#pragma omp parallel for` construct around the evolutionary loop that computes the new generation (see Listings 1.1 and 1.2). However, due to the irregular work load, one should not use the default *static* scheduler and should use instead *dynamic* or *guided*. Indeed, for our case *static* scheduler only performs a speedup up to 16 on 32 cores for the best configuration with lowest mutation rate. Table 1 reports the measured performance of this approach on a 16, 32 and 64-core machine against a sequential execution. By summing the duration d_i of each iteration, we are able to compute the work W of a complete generation[4] in order to express the *idle proportion* $I = 1 - \frac{W}{p \times T_p}$ where the p cores stay idle during the time T_p of the for-loop execution, with no iteration left to distribute and wait for others to finish their work. There are two important remarks. First, the work is inflated when running on parallel NUMA architecture [19]. That explains the poor efficiency with respect to a lower idle proportion. Second, the efficiency is not that good. It is important to realise that the number of iterations (the population size) is not that large

[4] Iterations only perform computation.

# cores	Mutation rate	SpeedUp Dynamic	SpeedUp Guided	Efficiency Dynamic	Efficiency Guided	Idle proportion Dynamic	Idle proportion Guided
16	10^{-4}	11.2	10.9	70%	68.1%	4.1%	7.5%
	10^{-5}	11.2	10.1	70%	63.1%	5.0%	14.6%
	10^{-6}	10.2	8.6	63.8%	53.8%	11.2%	27.0%
32	10^{-4}	20.3	19.5	63.4%	60.9%	9.2%	15.3%
	10^{-5}	20.0	17.0	62.5%	53.1%	11.1%	26.3%
	10^{-6}	16.5	13.1	51.6%	40.9%	21.4%	41.7%
64	10^{-4}	34.4	32.8	53.8%	51.3%	18.2%	27.0%
	10^{-5}	33.1	26.2	51.7%	40.9%	22.1%	41.1%
	10^{-6}	23.9	18.2	37.3%	28.4%	36.6%	56.5%

Table 1. Performance of Aevol with several mutation rates using *dynamic* and *guided* schedulers. Environment : gcc8.3, libGOMP on 4 SkyLake Xeon Gold 6130.

in respect with the number of cores. Moreover, there are even fewer iterations that represent the treatment of mutants: Function `compute` of listing 1.2 is very fast for non-mutants. This kind of irregularity makes the *guided* scheduler ineffective [18]. For the case of the *dynamic* scheduler, even with 64 cores, speed-up is only around 33. The idle proportion of the cores varies from 9% to 36%. This reveals work imbalance due to not so good schedule. The population size is a very important parameter for a biological point of view [2]. Doing experiments with small, medium or large population will not produce the same results and cannot be interpreted the same way. Thus, we cannot blindly increase the size of population to convey more parallelism because the experiments will not be the same.

3.2 Scheduling Iterations Based On Their Data

A finer inspection of the evolution loop of Aevol shows that it iterates over a *composition of functions* $f_n \circ ... f_2 \circ f_1$ applied to each individual (see Listing 1.3). The classical list-scheduling algorithms [10] such as implemented in OpenMP runtimes delivers medium level of performance with parallel efficiency ranging from 37% to 63% as shown in the previous section. To increase performance of the loop scheduler, we need extra information to schedule loop with a better clairvoyance. For instance, LPT [11] requires the knowledge of execution time for a better competitive ratio.

```
1 for i = 1..N do
2    fitness[i] = f_n ∘ ... f_2 ∘ f_1(indiv[i])
```

Listing 1.3. Initial evolutionary loop.

```
1 for i = 1..N do
2    r[i] = f_k ∘ ... f_2 ∘ f_1(indiv[i])
3 for i = 1..N do
4    fitness[i] = f_n ∘ ... f_{k+2} ∘ f_{k+1}(r[i])
```

Listing 1.4. Our loop decomposition.

Our approach is to look whether the data generated during the execution of a function f_k may give clues on the remaining computation of this iteration.

Accordingly, our methodology is to split the functions into two groups and to specialize the scheduling algorithm for each part: i) let schedule the first k function calls $f_k \circ ... f_2 \circ f_1$ with a non-clairvoyant scheduling algorithm, and ii) let use the data produced after this step to gain in information to better schedule the remaining function calls $f_n \circ ... \circ f_{k+1}$ by a clairvoyant scheduling algorithm.

Listing 1.4 illustrates the resulting loops after having split the loop in two. The remaining questions are: Why not to split in more than two the composition? Which kind of clairvoyant loop scheduling algorithm? and finally: How to find the right separator k to split the composition of functions? The first question is related to the structure of the computation and we show *a posteriori* that splitting the loop in two is enough. Moreover, each loop decomposition implies a synchronization and we have found that a good trade-off for this application is two.

Because the computation is at fine grain, we have decided to select an existing loop scheduler with low overhead at runtime. Our final choice was to base our second loop scheduler on the original LPT algorithm [11] where individuals are sorted according to the size of their DNA after mutation. This information strongly correlates to the execution time of a mutant and the next sections focus on it.

3.3 Predicting the Execution Time

A deep analysis of the Aevol code and, more precisely, the underlying computational biological model, leading to the creation of the Figure 1, was necessary. Knowing, for instance, that the `Translation` function take as input the list of all the genes of an individual, we could induce the time of this function with the number of genes *i.e.*, thanks to the output of the `Find_Genes` function. Going backward again, as genes are segments of characters inside RNA (which is also a segment of characters inside the total DNA), the more RNA will be found in the function `Find_RNA`, the more time function `Find_Genes` will take.

Finally, keeping the same logic, Figure 3 illustrates that the size of the DNA after mutation linearly correlates with the execution time of the iterations of the loop $f_n \circ ... f_2(r[i])$, where f_1 applies the mutations on one individual. The parameter of the linear model changes over the generations, but the linearity between size of DNA and execution time permits us to schedule loop with the LPT algorithm where individuals are sorted accordingly to their DNA sizes.

Simulation of LDNA schedule. To test our hypothesis, we collect the execution time of each function f_i call on all the individuals and we simulate our scheduler called LDNA: as for LPT individuals are sorted decreasingly with their DNA size after mutation. We compare our LDNA with respect to LPT thanks to the postmortem simulation with known execution times. Figure 4 displays the simulated efficiency for 100 generations with the *dynamic* scheduler of OpenMP (Dyn), LPT, and LDNA on a 64 cores machine. We see that LDNA almost achieves the performance of LPT which is almost optimal most of the time. Next sections present how to build LDNA for the evolution loop.

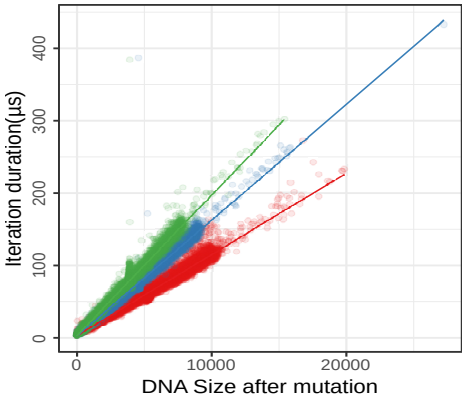

Fig. 3. Scatter plot with duration of iterations vs the size of DNA after mutation. The different color represent different point in time of the execution.

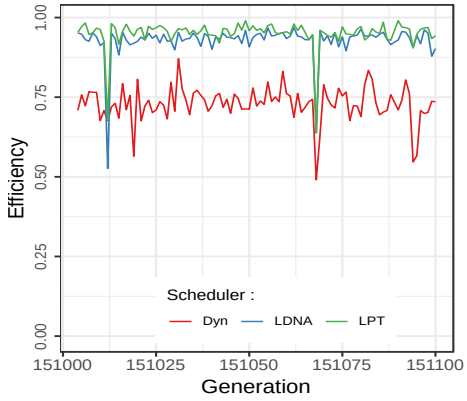

Fig. 4. Simulation of a posteriori schedules of generation of Aevol on 64 cores with different scheduler

```
1  #pragma omp parallel for schedule(static)
2  for (auto i = 0; i<N; ++i) {
3    indiv[i] = prepare_mutation ∘ selection(cell[i])
4    if has_mutate(indiv[i])
5      mutant_list.push_back(i) // Concurrent access to the list
6  }
7  << synchronize_sort(mutant_list) >>
8  #pragma omp parallel for schedule(monotonic: dynamic(1))
9  for (auto i: mutant_list)
10   fitness[i] = do_fitness ∘ ... ∘ do_mutation(indiv[i])
```

Listing 1.5. General structure of the code to compute a generation with LDNA. Sections of the code depend on the way sort is implemented, see Section 3.5

3.4 LDNA, A Scheduling Algorithm for Aevol

Listing 1.5 describes the new organization of the computation of Aevol with two parallel loops following our loop decomposition. The first loop computes in parallel the selection and prepares the mutations and give us the new DNA size of the mutants. It also discriminates the mutants inside a shared data structure. Because the duration of the iterations of this loop are small and with less irregularity than the initial problem, we can apply a static scheduling. At this point, the computation for the non-mutants is finished and we only have to deal with mutants. As previously, we applied the simplicity of LPT schedule with the DNA size of the mutant. Iterations are sorted with this data by `synchronize_sort()` as shown in the listing 1.5. This function hides the complexity of managing the list of mutants which is a data structure shared by all the working threads. Multiple implementations of this list are discussed in the next section. Finally, the

second loop is executed in parallel using the LPT rule with the mutant list as iteration space. The next section will present how we implement the LPT rule with OpenMP.

3.5 OpenMP Implementation of LDNA

The LPT rule is originally an off-line scheduling technique. Therefore, if one wants to implement it with OpenMP as an off-line scheduler, one must touch the OpenMP runtime. However, because our goal was to apply our solution without touching to the OpenMP runtime, we used the dynamic schedule with 1 iteration per chunks using an already sorted list for the iteration space as shown in Listing 1.5. This configuration will complete an LPT schedule if we assure that an idle thread will pick the next iteration on the logical order, *i.e.*, the longest available iteration. Fortunately, OpenMP4.5 [20] introduces the *monotonic* modifier to be added to the scheduler (as shown on the second loop): it ensures that chunks are assigned in the increasing logical iteration order.

An other issue was the management of the list of mutants shared between all threads and subject to concurrent accesses. Because of fine grain operation, the best implementation is a compromise between an algorithmic variant and the overhead at runtime. We have followed a pragmatic experimental evaluation of several variants that relies on different OpenMP features to manage it.

Our two promising implementations distribute the list where each thread keeps a local sorted list, then `synchronization_sort` (Line 7) merges all the data. In our first implementation, the merge operation is be done in two ways: i) using the `reduction` construct of OpenMP4.0 or ii) do it ourselves. The first implementation views concurrent list insertion as reduction operation between lists. It relies on the declaration of reduction operator, called by the OpenMP runtime, in charge of merge two lists. We call this implementation `LDNA_Omp_Redux`.

In our second implementation, all the local lists are merged by our program using a binary merge tree. This is `LDNA_Par_Tree`. Parallel merge may be of interest but it depends on the size of the list. A first attempt has shown that parallelism variant does not outperform sequential binary merge with useful data size for our problem. So we call this implementation `LDNA_Seq_Tree`.

omp for vs taskloop The OpenMP standard propose another way of parallelism using tasks. The *taskloop* construct allows to execute and schedule chunks of iterations as tasks. One could even turn each sub function of an iteration in a new task and the program could convey more parallelism. However, the current implementations of OpenMP have tremendous overhead at the creation of tasks[9] prohibiting their use in the case of lots of small tasks. This is why we only use the *omp for* construct for our implementation waiting for evolution in the tasks management by the OpenMP runtimes.

4 Experimental Results

This section deals with the evaluation of LDNA against the *dynamic* schedule of OpenMP. For the experimentation, the program was compiled with GCC 8.3, linked with jemalloc5.2.1 [7] as a memory allocator more suited for parallel allocations. The OpenMP runtime is libGOMP and the execution was done on a *yeti* node running on Debian 10 from the Grid5000 platform. A node is equipped, with 4 Skylake Intel Xeon Gold 6130 processors for a total of 64 cores (Hyper-Threading was not used) and 768 GiB of memory on 4 NUMA sockets. The memory allocation policy is the *first-touch* policy. As the computation of a mutant asks the thread to copy (meaning memory allocation) the ancestor and then work on the copy a thread will then work on his local NUMA node to process a mutant. We select the libGOMP runtime because other runtimes (from LLVM or Intel), did not show significant difference in performance.

4.1 Protocol of Experimentation

All the experiments were populated with 1024 individuals (see reason in section 2.1). Three mutation rates are used: 10^{-4}, 10^{-5} and 10^{-6}. For each mutation rate, we did 4 repetitions with different seeds for the random generator. For each experiment, we selected 6 starting generations separated by 50,000 generations from Generation 1000 to Generation 251,000. For each the 72 starting generations, the protocol of execution was the following: for doubling numbers of cores from 4 to 64 (using the least NUMA nodes possible thanks to *numactl*), we computed 100 generations in which we timestamped the beginning and the end of each iteration. During our preliminary study of Aevol, we observed that the behavior of the computation only changes on large scale of generations. This explains why we take this few contiguous generations but spread on 251,000 generations.

4.2 Results

The following figures summarize data averaged on all the executions. We observed that the first 3 generations had a strange behavior certainly due to a warming up effect and they are not counted in these means. Figure 5 compares the proportion of the time taken by the `synchronization_sort` step during one generation. `LDNA_Seq_Tree` shows itself the best option over the two solutions. For the case of `LDNA_Omp_Redux`, the results were surprising and further analysis shows that all the operations of reduction occur sequentially with lots of time spent in the OpenMP runtime. We chose to continue experiments with `LDNA_Seq_Tree` as our `LDNA` scheduler.

Figure 6 compares the speedup of our `LDNA` scheduler with the reference *dynamic* scheduler of OpenMP (`Dyn`). It is clear that `LDNA` outperforms `Dyn`. For 64 cores the `LDNA` scheduler is on average 19%, 21% and 27% faster for respectively 10^{-4}, 10^{-5} and 10^{-6} mutation rates. In Figure 7, an example of the execution of one generation with mutation rate 10^{-5} is given. It shows how `LDNA`

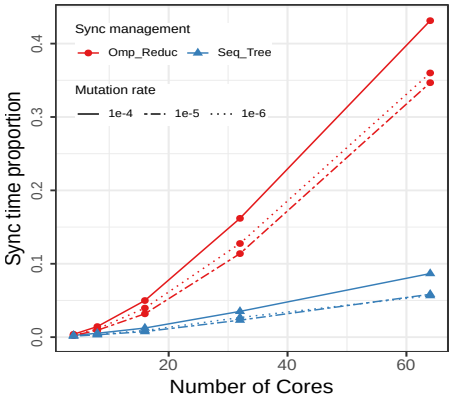

Fig. 5. Average proportion of time used for synchronization with the different implementations depending on mutation rate

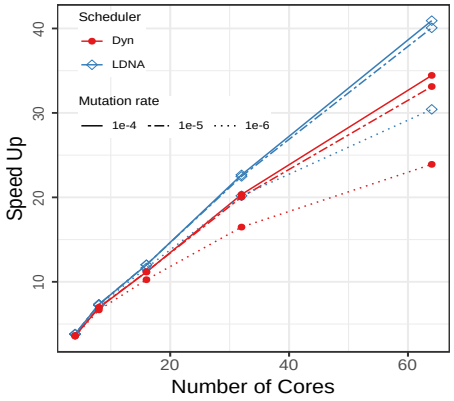

Fig. 6. Average speed up of `Dyn` and `LDNA` depending on the mutation rate

(bottom) succeeds to compact the computation compared to `Dyn` (top). With the latter, the mutants (blue) and non-mutants (red) are treated in a random order, explaining why large mutants are computed only at the and. For `LDNA`, non-mutants are all treated in the first part which is scheduled statically. This reduces the schedule overhead that the non-mutants induce compared to the `Dyn` scheduling. The second part only computes what remains for the mutant in a specific order that permits to compact all the iteration. The blank part corresponds to the sort and merge of the list of mutants. For this example, it takes about 0.03 ms which represents 5% of the computation time (On average for a mutation rate at 10^{-5}, it is 5.8%, figure 5).

For 10^{-4} and 10^{-5} mutation rates, the idle time proportion, without counting synchronization and sort, dropped to a maximum of 7%. In the case of 10^{-6} mutation rate, we can see that it scales less than the others, and the idle time is more difficult to reduce. In fact, the mutation rate is so low that the number of mutants reaches is about 60 individuals. Therefore, the number of iterations is very close to the number of cores and sometimes less. The program lacks of parallelism and a solution would be to parallelize at the sub-functions grain. As seen in section 3.5 this could be easily done with task parallelism but will suffer the large overhead due to the current OpenMP runtime.

4.3 Evaluation on Larger Populations

If the size of the population is 10 times larger than in previous experiments, and future use of Aevol could use this for biological interests, the parallelism would be greater. As the number of iterations rises, a simple dynamic schedule could be enough and `LDNA` could suffer from the overhead from the management of the mutant list. Nevertheless, `LDNA` succeeds to scale up the population better

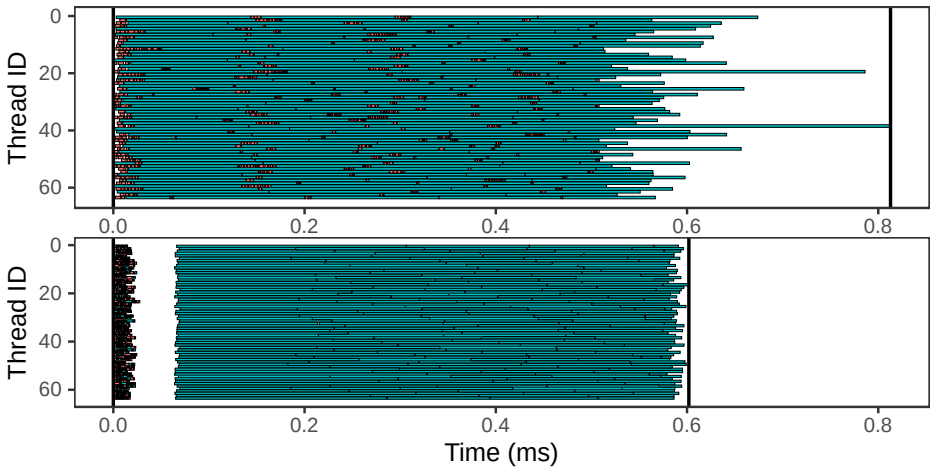

Fig. 7. Example of the execution of one generation with two scheduler. Blue rectangles correspond to mutants and red to non-mutants. At the top `Dyn`, where iteration executes at random order, and at the bottom `LDNA`, where the few step of evolution are performed with a *static* scheduler, then the mutant list is merged and sorted (blank part) and then only mutants are evaluated to find their fitness.

than `Dyn`. Larger experiments with 9.216 individuals on 64 cores show that our scheduling stay better but not as much. The difference is around 12% with mutation rate at 10^{-6} and 2% at 10^{-4}. With the smaller mutation rate, mutants are enough so that `Dyn` suffers from the irregularity. At the end, a larger population would mean that any scheduling algorithm would approach optimal result but simulations with small population will still be used in the future. However, to avoid prohibitive computation times, in the case of a larger population, we will have to use a larger number of cores and thus return to a similar scheduling problem that the one with 1.024 individuals on 64 cores. Still, LDNA could do better because we observe that NUMA effects became important and it is clear that the cores wait because of communication latency during the first loop. Taking NUMA into account would certainly improve LDNA scalability.

5 Related Work

Our scheduling problem is largely studied since at least 50 years. In [12], the $P||C_{max}$ problem is presented as NP-complete. Two approaches exist to deal with this problem: whether the duration of iteration is known in advance or not. [14] is certainly the best off-line algorithm but LPT [11] is a well known heuristic with great performance [5] and simplicity and an even better version has been developed by Cheng et al. [4].

When the information about the iterations is unknown before execution, the *list-scheduling* algorithm described by Graham [10] is the upper-bound limit and

the basic technique for most of the dynamic scheduler developed since. After an initial static distribution of the iterations, Durand et al. [6] implemented a work-stealing method with memory location awareness. Lucco [18] presented a guided self-scheduling scheme improved with statistics computed on early iterations. In overall, the idea is to find information during execution to refine the scheduling. Besides an approach applied in [21,22] consists on letting the user provides workload estimation of the iterations before execution to perform near-optimal static scheduling and balance the final workload with dynamic work-stealing to catch up the possible mistakes of the estimation. Our approach is similar but cannot use user-provided estimation because of the stochasticity of Aevol. Instead, it has to use estimation from the application itself and these estimation change generation after generation. Our method cannot be easily embedded into an OpenMP loop scheduler because it requires (manual) loop decomposition and analysis of the structure of the application. At this expense, we are able to improve existing loop OpenMP schedulers with up to 27% on fine grain loop.

6 Conclusion and Future Work

In this paper, we present a new methodology to schedule irregular independent iterations of an application structured as a composition of functions. Mixing non-clairvoyant and clairvoyant techniques, we show that splitting the execution of the loop in several loops is a valid approach if the data gathered in the first loop help the scheduling of the next one. Applying our method to Aevol, a computational biology software, we implemented the algorithm `LDNA` to schedule the computation of evolving uni-cellular organisms. Experimental evaluations on a multi-core architecture computer show that our scheduler improves by about 27% the performance of the dynamic scheduler of OpenMP often used for irregular applications. We discuss on the implementation of the method and how to manage the synchronization to optimize the execution of our solution.

As other work [22] that uses workload-aware scheduler, we think that allowing the user to inform the runtime through the OpenMP standard would help this kind of method. The standard could accept an estimation function or even a way to pass a scheduler to be used by the runtime.

The management of the list of mutants brought to use a binary merge tree that we kept sequential. However, this part take up to 10% of the total computing and more work could certainly find a way to parallelize efficiently this part.

Finally, it would be on interest to evaluate if the methodology used to find where to split the evolutionary loop could be generalized and automated to other parallel applications.

Acknowledgement

Experiments presented in this paper were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS,

RENATER and several Universities as well as other organizations (see https://www.grid5000.fr).

References

1. Banicescu, I., Velusamy, V.: Load balancing highly irregular computations with the adaptive factoring. In: Proceedings 16th International Parallel and Distributed Processing Symposium. pp. 12 pp– (Apr 2002)
2. Caballero, A.: Developments in the prediction of effective population size. Heredity **73**(6), 657–679 (Dec 1994), number: 6 Publisher: Nature Publishing Group
3. Card, K.J., LaBar, T., Gomez, J.B., Lenski, R.E.: Historical contingency in the evolution of antibiotic resistance after decades of relaxed selection. PLoS biology **17**(10) (2019)
4. Cheng, T.C.E., Kellerer, H., Kotov, V.: Algorithms better than LPT for semi-online scheduling with decreasing processing times. Operations Research Letters **40**(5), 349–352 (Sep 2012)
5. Coffman, Jr., E.G., Sethi, R.: A Generalized Bound on LPT Sequencing. In: Proceedings of the 1976 ACM SIGMETRICS Conference on Computer Performance Modeling Measurement and Evaluation. pp. 306–310. ACM (1976)
6. Durand, M., Broquedis, F., Gautier, T., Raffin, B.: An Efficient OpenMP Loop Scheduler for Irregular Applications on Large-Scale NUMA Machines. In: Rendell, A.P., Chapman, B.M., Müller, M.S. (eds.) OpenMP in the Era of Low Power Devices and Accelerators. pp. 141–155. Springer, Berlin, Heidelberg (2013)
7. Evans, J.: A Scalable Concurrent malloc(3) Implementation for FreeBSD p. 14 (Apr 2006)
8. Fischer, S., Bernard, S., Beslon, G., Knibbe, C.: A model for genome size evolution. Bulletin of mathematical biology **76**(9), 2249–2291 (2014)
9. Gautier, T., Pérez, C., Richard, J.: On the Impact of OpenMP Task Granularity. In: IWOMP 2018 - 14th International Workshop on OpenMP for Evolving Architectures. pp. 205–221. Springer, Barcelone, Spain (Sep 2018)
10. Graham, R.L.: Bounds for Certain Multiprocessing Anomalies. Bell System Technical Journal **45**(9), 1563–1581 (1966)
11. Graham, R.L.: Bounds on Multiprocessing Timing Anomalies. SIAM Journal on Applied Mathematics **17**(2), 416–429 (1969)
12. Graham, R.L., Lawler, E.L., Lenstra, J.K., Kan, A.H.G.R.: Optimization and Approximation in Deterministic Sequencing and Scheduling: a Survey. In: Hammer, P.L., Johnson, E.L., Korte, B.H. (eds.) Annals of Discrete Mathematics, vol. 5, pp. 287–326. Elsevier (Jan 1979)
13. Hindré, T., Knibbe, C., Beslon, G., Schneider, D.: New insights into bacterial adaptation through in vivo and in silico experimental evolution. Nature Reviews Microbiology **10**(5), 352–365 (2012)
14. Hochbaum, D.S., Shmoys, D.B.: Using Dual Approximation Algorithms for Scheduling Problems Theoretical and Practical Results. J. ACM **34**(1), 144–162 (Jan 1987)
15. Knibbe, C.: Structuration des génomes par sélection indirecte de la variabilité mutationnelle : une approche de modélisation et de simulation. thesis, Lyon, INSA (Jan 2006)
16. Liard, V., Parsons, D., Rouzaud-Cornabas, J., Beslon, G.: The Complexity Ratchet: Stronger than selection, weaker than robustness. Artificial Life Conference Proceedings **30**, 250–257 (Jul 2018)

17. Liard, V., Rouzaud-Cornabas, J., Comte, N., Beslon, G.: A 4-base model for the aevol in-silico experimental evolution platform. In: Knibbe, C., Beslon, G., Parsons, D.P., Misevic, D., Rouzaud-Cornabas, J., Bredèche, N., Hassas, S., Simonin, O., Soula, H. (eds.) Proceedings of the Fourteenth European Conference Artificial Life, ECAL 2017, Lyon, France, September 4-8, 2017. pp. 265–266. MIT Press (2017)
18. Lucco, S.: A dynamic scheduling method for irregular parallel programs. In: Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation. pp. 200–211. Association for Computing Machinery, San Francisco, California, USA (Jul 1992)
19. Olivier, S., Supinski, B., Schulz, M., Prins, J.: Characterizing and mitigating work time inflation in task parallel programs. vol. 21, pp. 1–12 (11 2012)
20. OpenMP Architecture Review Board: OpenMP Application Program Interface. Specification (2015), https://www.openmp.org/wp-content/uploads/openmp-4.5.pdf
21. Penna, P.H., Castro, M., Freitas, H.C., Broquedis, F., Méhaut, J.F.: Design methodology for workload-aware loop scheduling strategies based on genetic algorithm and simulation. Concurrency and Computation: Practice and Experience **29**(22), e3933 (Nov 2017)
22. Penna, P.H., Gomes, A.T.A., Castro, M., Plentz, P.D.M., Freitas, H.C., Broquedis, F., Méhaut, J.F.: A comprehensive performance evaluation of the BinLPT workload-aware loop scheduler. Concurrency and Computation: Practice and Experience **31**(18), e5170 (2019)