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

T. Gautier, C. Perez, J. Rouzaud-Cornabas, L. Turpin

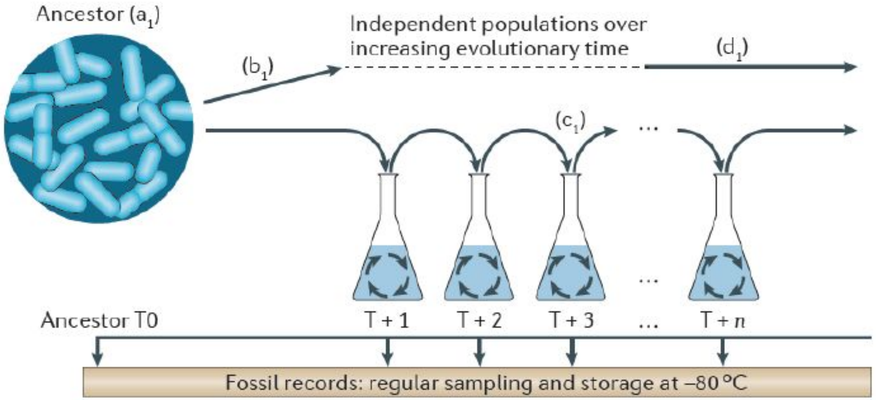
16th International Workshop on OpenMP, IWOMP 2020
Austin, TX, USA, September 22–24, 2020

A Field in Biology: Experimental Evolution

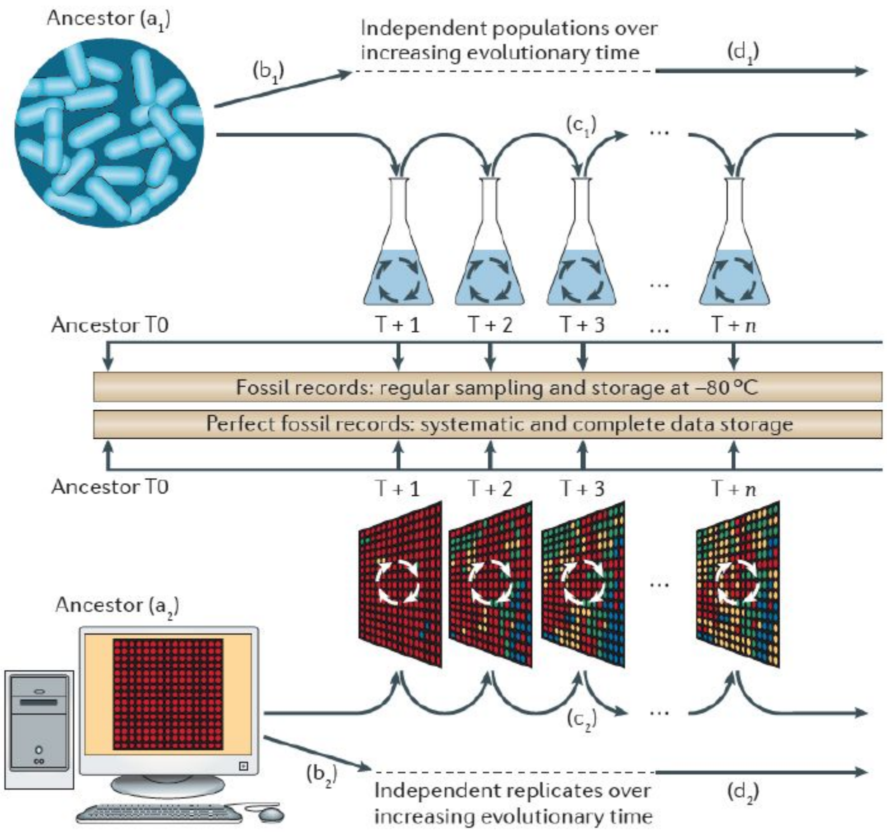
Long Term Evolution Experiment

- Richard Lenski (MSU/Beacon Center, USA)
- running since 1988
- 73,500 generations

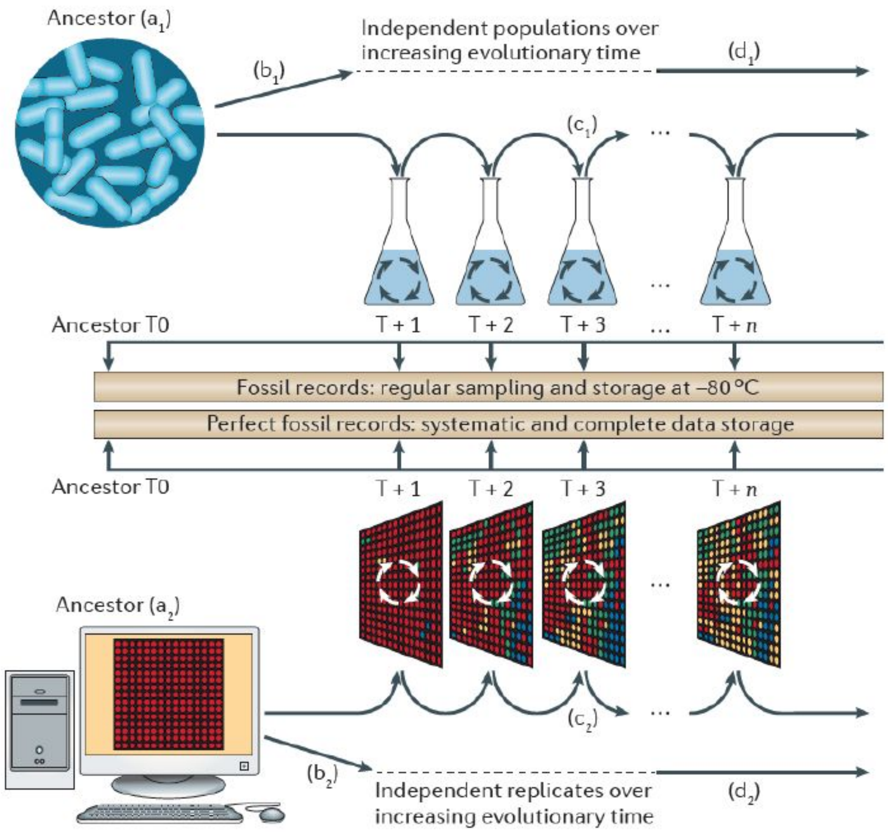
From *in-vitro* to *in-silico*



From *in-vitro* to *in-silico*



From *in-vitro* to *in-silico*

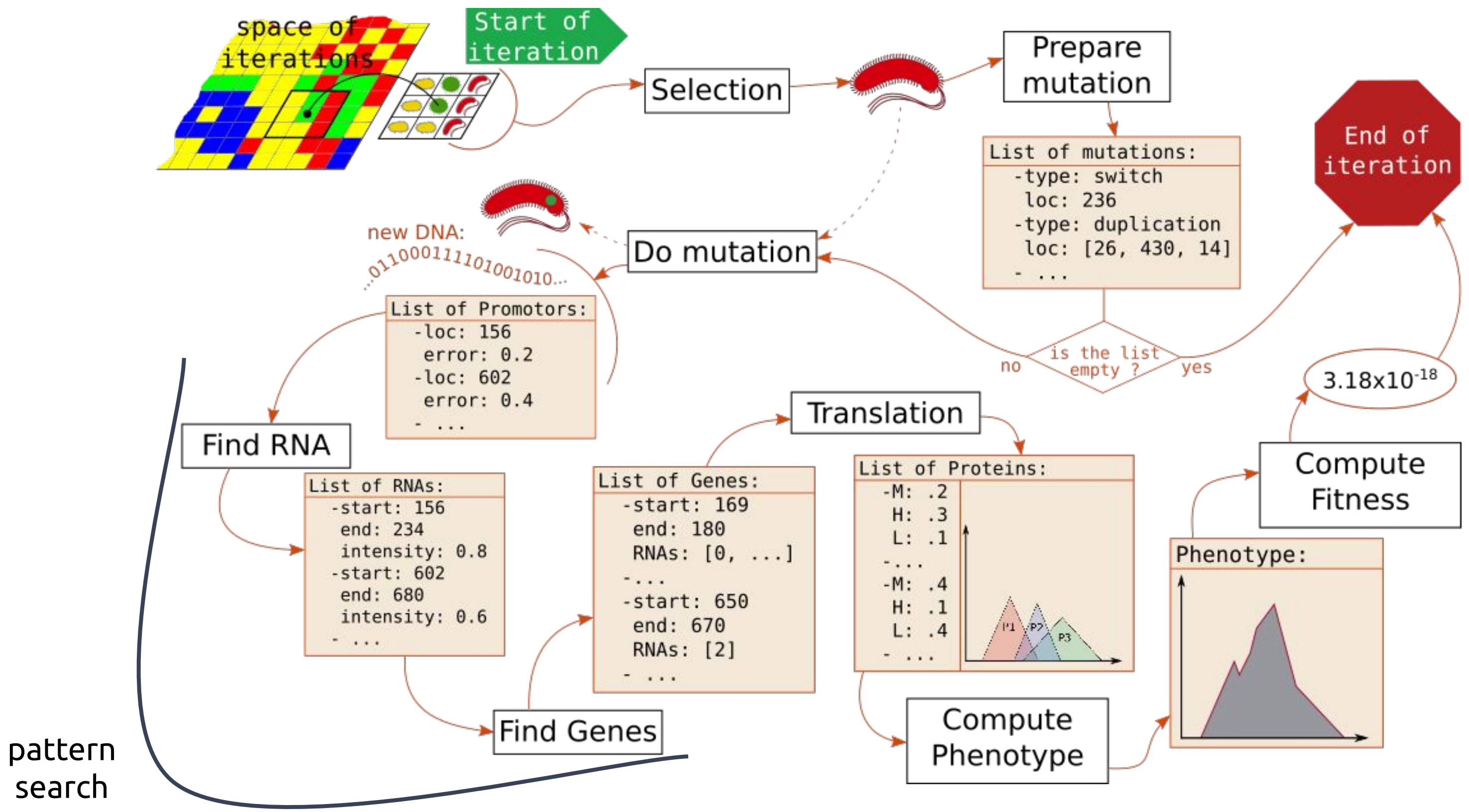


available at
aevol.fr

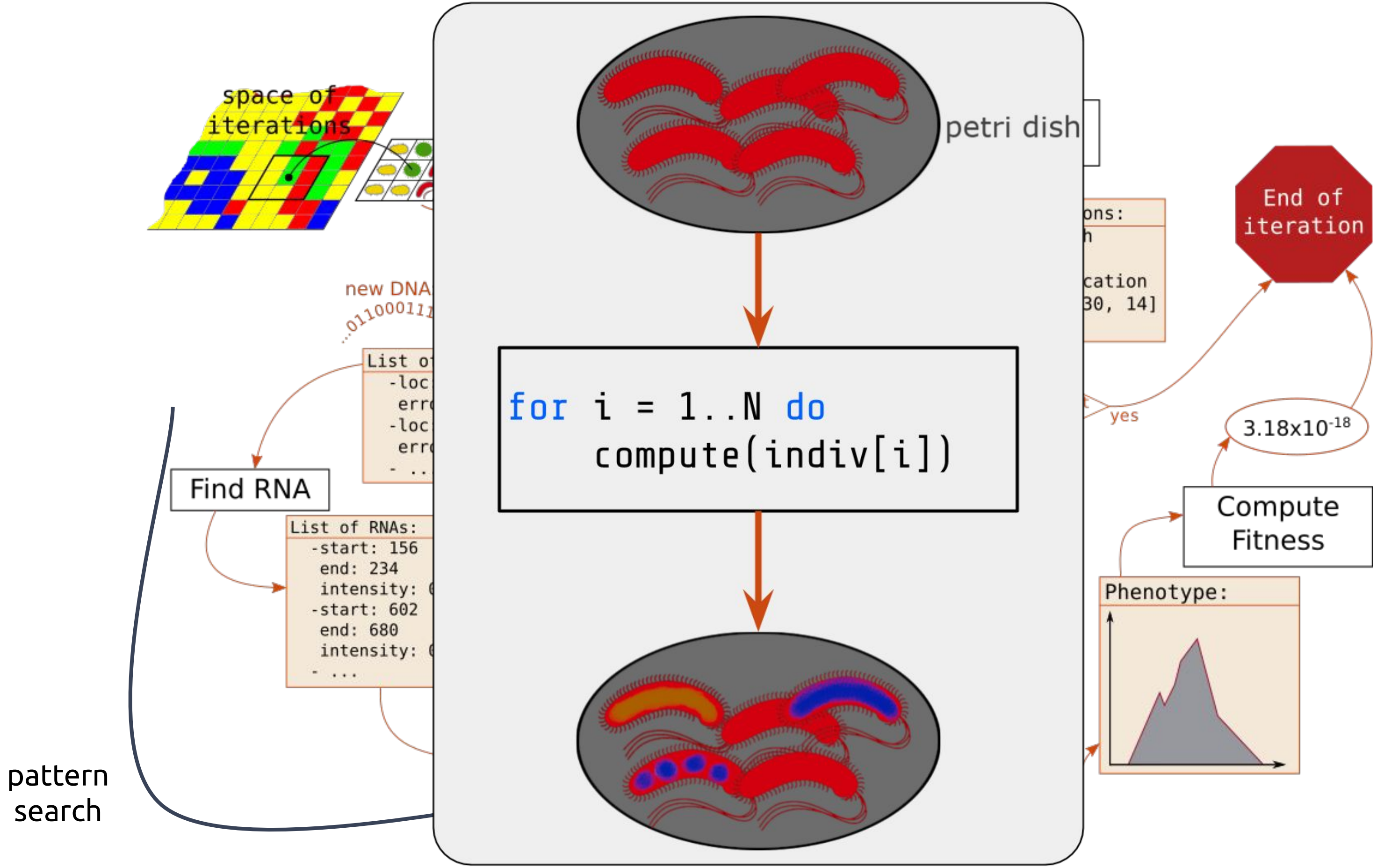
A biological simulator

- Around 50,000 C++ LOC on mono-node arch
- Simulate the evolution of micro organisms
- Compute sequentially one generation after the other
- For one experiment:
  - thousands of hours of computation
  - Terabytes of data (not I/O intensive though)
- Simulate millions of generations
  - around 30ms per generation (1024 individuals)
- **Goal** : Accelerate the computation of a generation

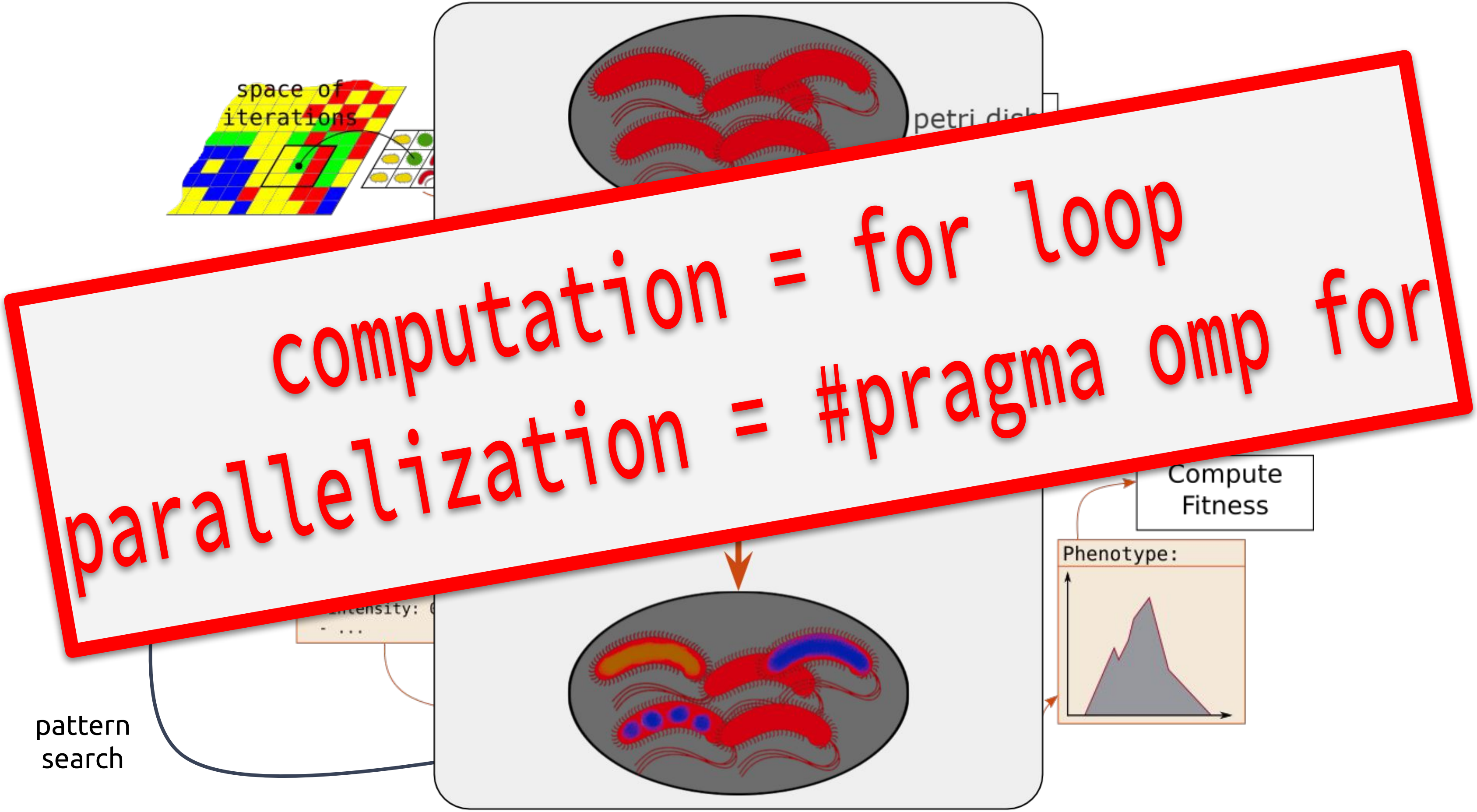
AEVOL: Workflow of a Generation



AEVOL: Workflow of a Generation



AEVOL: Workflow of a Generation



Disappointing Performance

How?

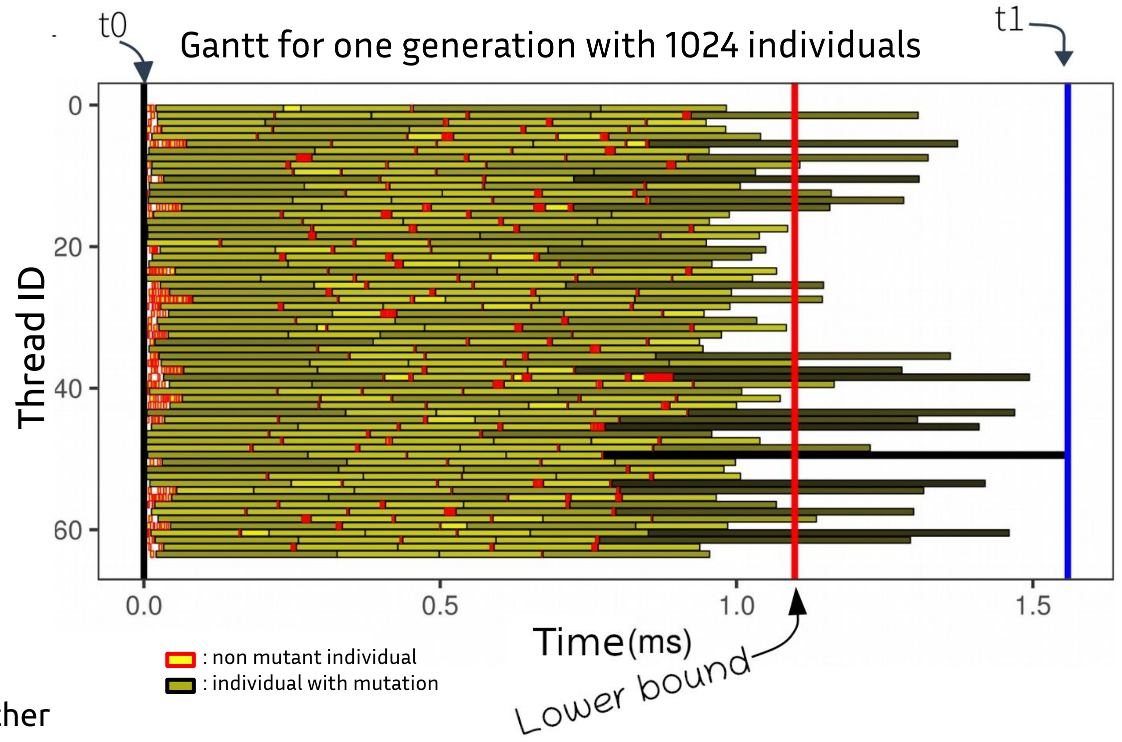
- OpenMP // loop + dynamic schedule
- GCC/libGOMP 8.3
- 64 core single node machine

Result

- More than 20% of idle time
- Speed up less than 36 on 64 cores
- Worse results with other OMP schedulers
  - Static or Guided

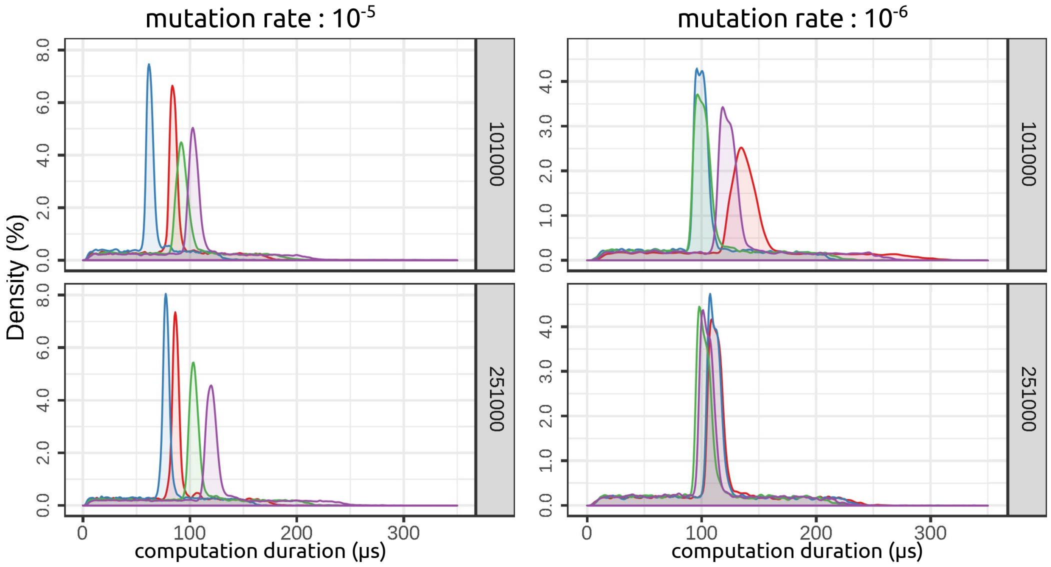
Why?

- Individual computation times
  - **irregular** (from 1 μs to 1,000 μs)
  - **can vary** from one generation to the other
  - **unpredictable** (stochastic simulation)



Characterization of the irregularity

Distribution of the computation time of mutants



Generation #101000

Generation #251000

Multiple colors stand for multiple experiments with different random number generator seeds

Mutations

- Stochastics events on the DNA (char array)
- Probability of occurrence depends on the **mutation rate** input argument and the current size of the DNA

2 Types of Population

- Non-Mutants
  - Take only 1% of the computation time
- Mutants
  - Take 99% of the computation time

Our Problematic

- Goal
  - Improve performance of irregular, varying, and unpredictable iterations
- Scheduling challenges
  - Reduce idle time and work inflation
- How to tackle it?

Related Work

Scheduling independent iterations

- List scheduling [Graham 1966]: Dynamic
- Base on bin-packing [Hochbaum & Shmoys 1987]
  - Complexe to implement outside the runtime
- LPT [Graham 1969]
  - Simplicity and Robustness [Coffman & Seti 1976]

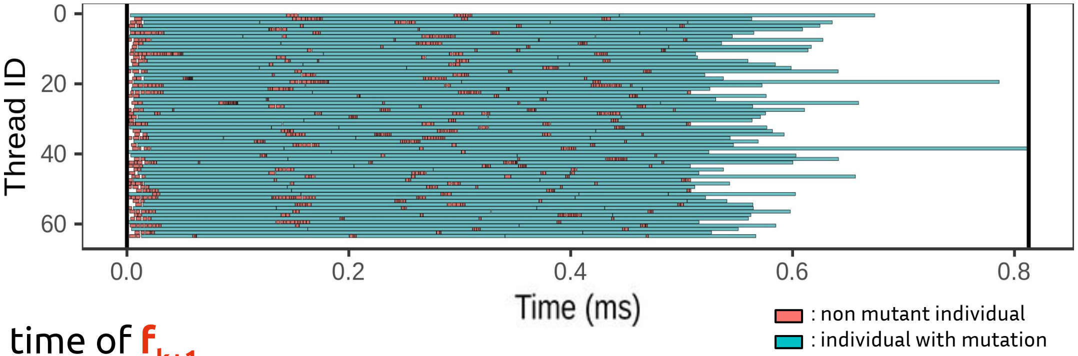
OpenMP loop scheduler

- Internal modification of an OpenMP runtime (libGOMP) [Durand 2013, ...]
- Passing information from application to OpenMP loop scheduler [Penna 2019]
  - Authors promote application with almost constant workload

Loop regularisation through decomposition

```
1  #pragma omp for schedule(dynamic, 1)
2  for i = 1..N do
3      fitness[i] = f_n ∘ ...f_2 ∘ f_1(indiv[i])
```

Speed up ~ 33

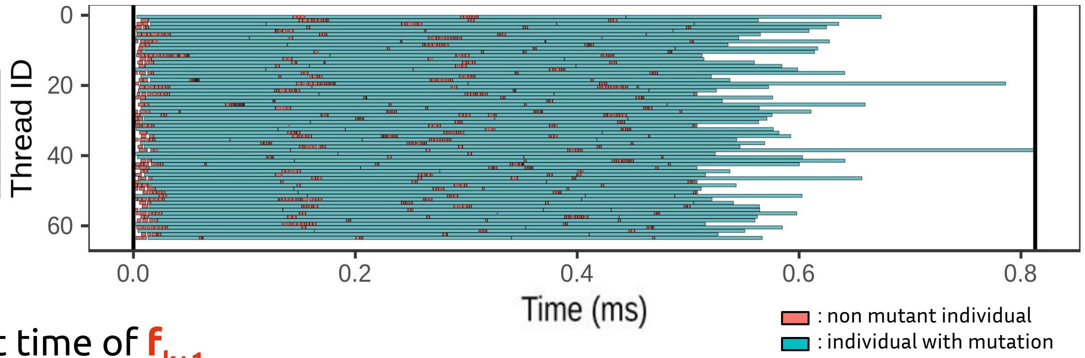


Suppositions

- Output from f_k may predict time of f_{k+1}
- Output from f_k may predict time from f_{k+1} to f_{k+j}

Loop regularisation through decomposition

```
#pragma omp for schedule(dynamic, 1)
for i = 1..N do
    fitness[i] = f_n ∘ ... f_2 ∘ f_1(indiv[i])
```



Speed up ~ 33

Suppositions

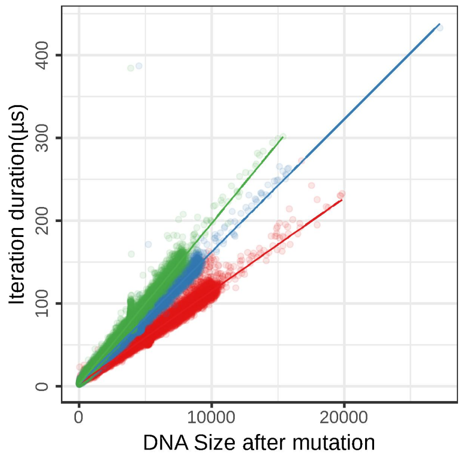
- Output from f_k may predict time of f_{k+1}
- Output from f_k may predict time from f_{k+1} to f_{k+j}

```
for i = 1..N do [in parallel]
    r[i] = f_k ∘ ... f_2 ∘ f_1(indiv[i])
schedule = compute_schedule(r)
for i = 1..N do [in parallel with schedule]
    fitness[i] = f_n ∘ ... f_{k+2} ∘ f_{k+1}(r[i])
```

Decomposition in 2 loops

- 1st loop to predict the load for the iterations of the 2nd loop
- Compute a schedule based on the prediction with a more clairvoyant strategy
    - LPT (Graham, 1966)
- 2nd loop is executed with the previously computed schedule

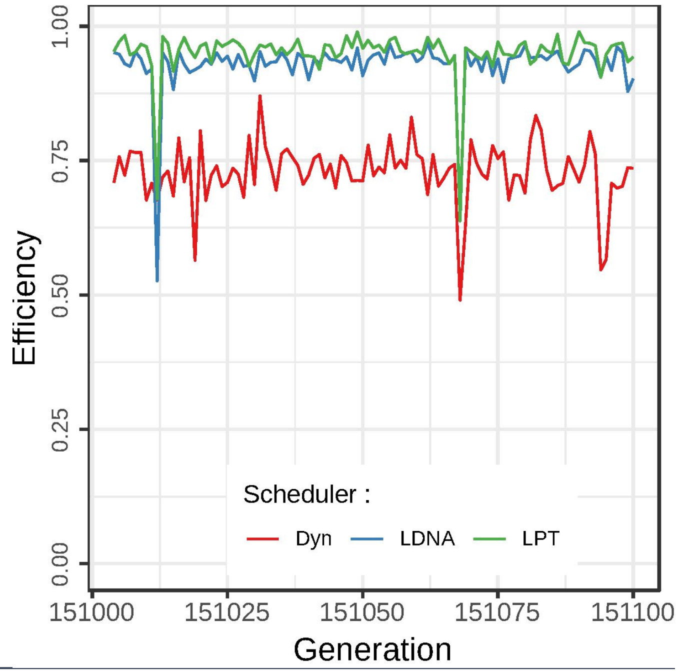
One data to explain them all



colors correspond to
different generation

Case of Aevol

- Linear relation between duration and size of DNA after mutation
- Linear relation depends on the generation
- Sufficient for an LPT schedule generation after generation
- Let’s call it LDNA



Confirmation by Postmortem Simulation

- Comparing three scheduling strategies
  - (Non clairvoyant) Dynamic and LDNA
  - Clairvoyant LPT
- LDNA almost as good as LPT

Sketch of the final solution

```
vector<Mutant> mutant_list; // In global scope
#pragma omp parallel
{
#pragma omp for schedule(static)
for (auto i = 0; i<N; ++i) {
  indiv[i] = prepare_mutation ∘ selection(cell[i])
  if has_mutate(indiv[i])
    mutant_list.push_back(i) // Concurrent access to the list
}
<< synchronize_sort(mutant_list) >> // Sorted by new DNA size
#pragma omp for schedule(monotonic: dynamic(1))
for (auto i: mutant_list)
  fitness[i] = do_fitness ∘ ... ∘ do_mutation(indiv[i])
}
```

Purely based on OpenMP Standard

- LPT thanks to dynamic scheduler
- `monotonic` modifier since OpenMP 4.5

Remaining issue

- Handle the list of mutants
- Efficient sort

Concurrent List of Mutants

Omp_Reduc

```
#pragma omp declare\
    reduction(merge: vector<Mutant>:\
              sort_merge_lists(omp_out, omp_in))
#pragma omp parallel
{
#pragma omp for reduction(merge: mutant_list)
for (auto i = 0; i < N; i++)
{... mutant_list.push_back(i) ...}
#pragma omp for schedule(monotonic: dynamic, 1)
for (auto i: mutant_list) {...}
}
```

VS

DIY

```
#pragma omp parallel
{
#pragma omp for nowait
for (auto i = 0; i < N; i++)
{... local_mutant_list[p_id].push_back(i) ...}
sort(local_mutant_list[p_id],
     [](m1, m2){ return size(m1) > size(m2); });
#pragma omp barrier
#pragma omp single
mutant_list = merge_lists(local_mutant_list)

#pragma omp for schedule(monotonic: dynamic, 1)
for (auto i: mutant_list) {...}
}
```

Concurrent List of Mutants

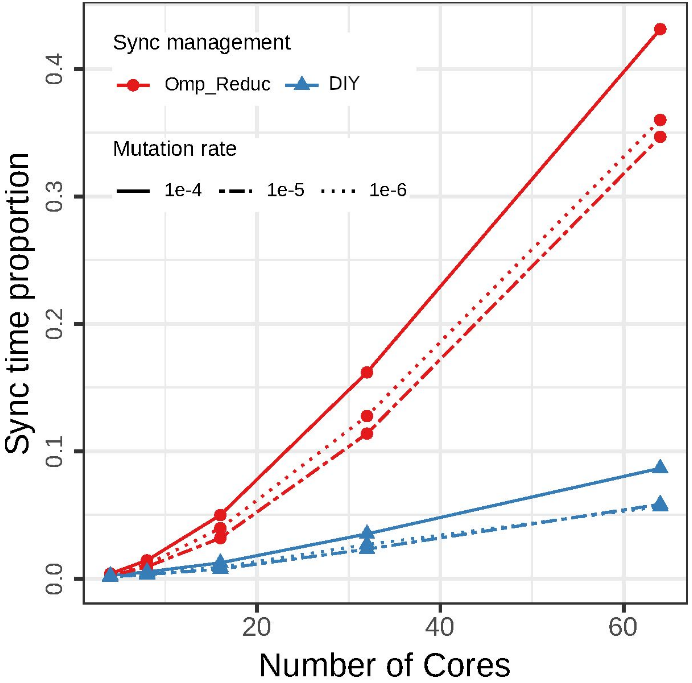
Omp_Reduc

```
#pragma omp declare\
    reduction(merge: vector<Mutant>:\
              sort_merge_lists(omp_out, omp_in))
#pragma omp parallel
{
#pragma omp for reduction(merge: mutant_list)
for (auto i = 0; i < N; i++)
{... mutant_list.push_back(i) ...}
#pragma omp for schedule(monotonic: dynamic, 1)
for (auto i: mutant_list) {...}
}
```

VS

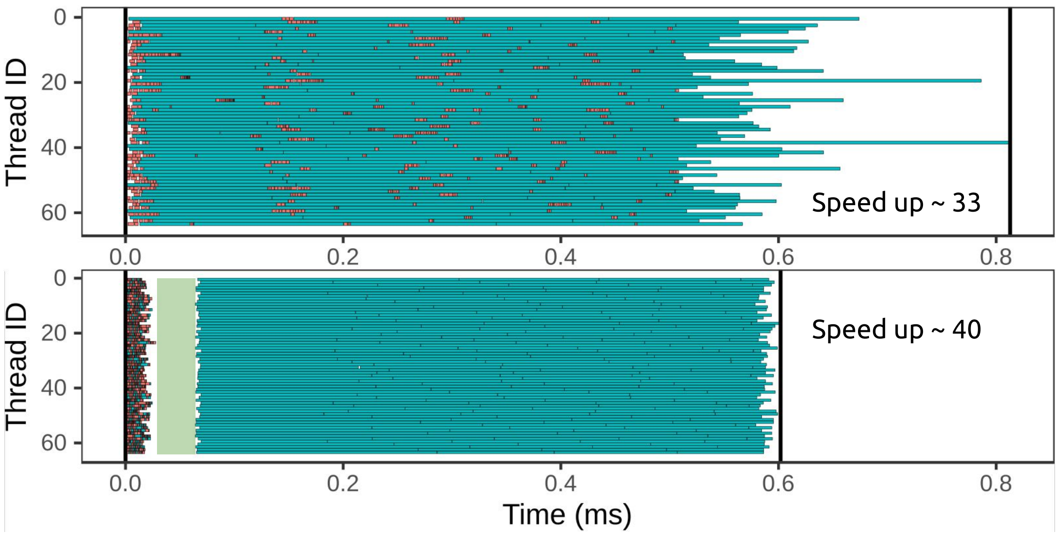
DIY

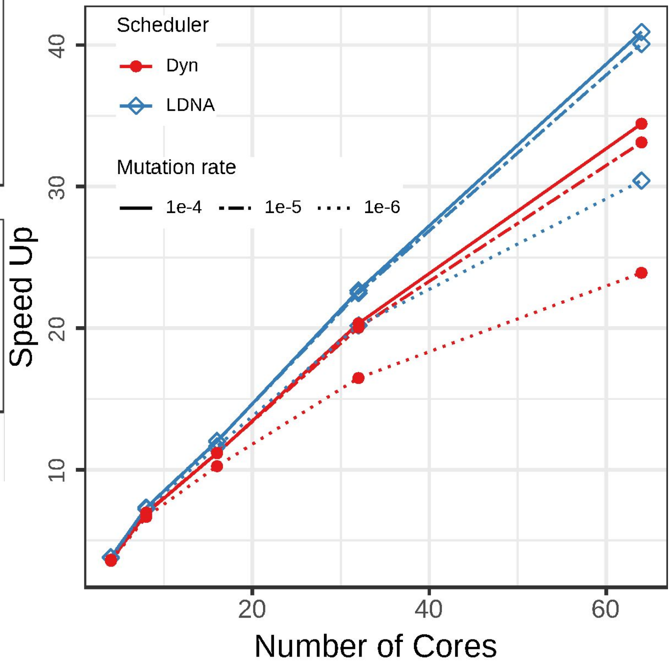
```
#pragma omp parallel
{
#pragma omp for nowait
for (auto i = 0; i < N; i++)
{... local_mutant_list[p_id].push_back(i) ...}
sort(local_mutant_list[p_id],
     [](m1, m2){ return size(m1) > size(m2); });
#pragma omp barrier
#pragma omp single
mutant_list = merge_lists(local_mutant_list)

#pragma omp for schedule(monotonic: dynamic, 1)
for (auto i: mutant_list) {...}
}
```



Proportion of time taken for the synchronization in one generation

Final Results





LDNA

- Solution purely based on OpenMP Standard
- ~ 20% of gain over Dynamic scheduler
- Ad Hoc solution for Aevol
  - Mix between biological model and parallelization model : no separation of concerns

Conclusion

AEVOL: Original computation pattern

- Highly irregular and varying application with unpredictable behavior over generation
- Fine grain computation
- Manual decomposition in two loops with specific scheduler
    - ~**20% of improvement**
- Need to analyse the biological model and its implementation to find an efficient solution to schedule the application

More realistic biological simulations need much more computation

Revisit Aevol parallelization to improve its performance

- Target multi-CPU/GPU node

OpenMP conclusion & perspective

AEVOL: Original/Unique computation pattern?

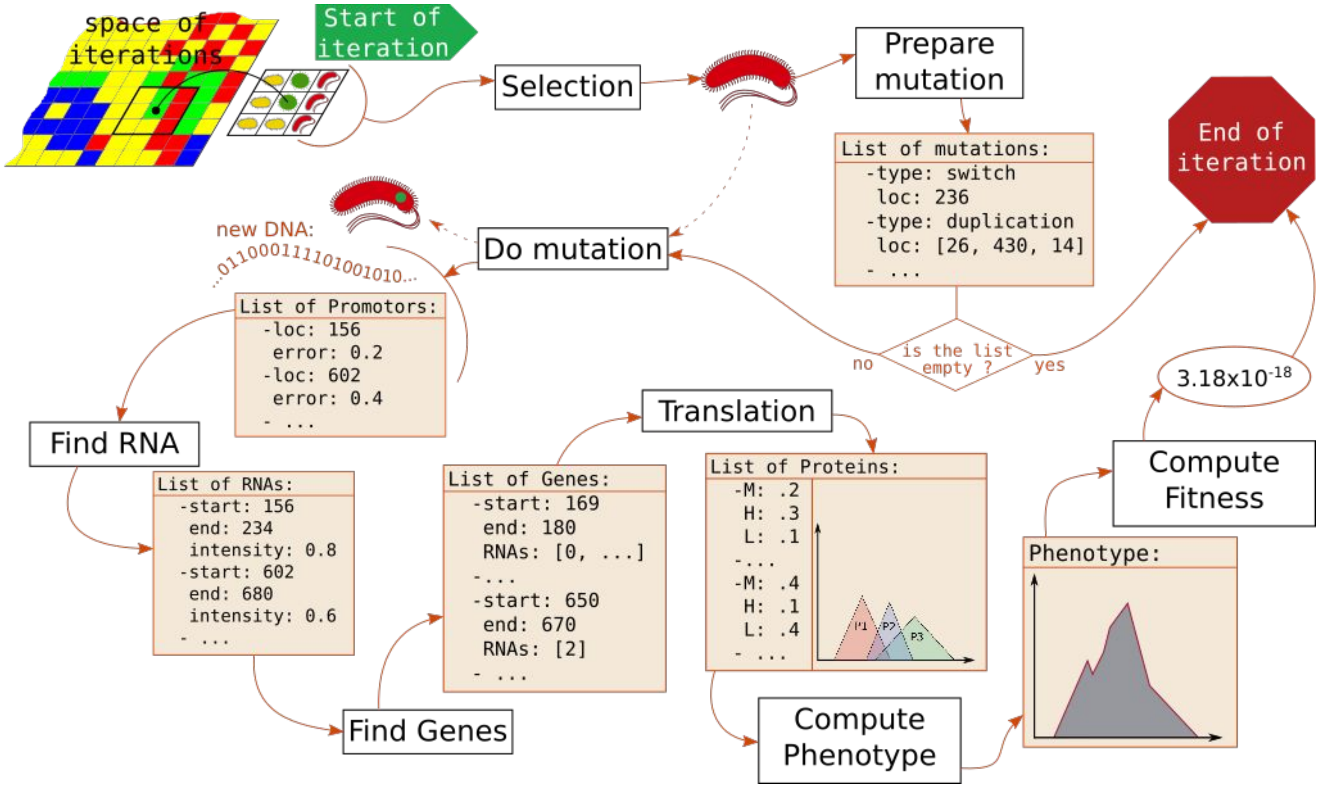
- Solution for CPU based purely on OpenMP standard
  - Through decomposition in 2 OpenMP for loops + specific LPT schedule
- **Code transformation depends on the scheduling solution!**
  - How to implement application dependent loop scheduler with code annotation only?
  - Have more clairvoyant schedulers in the OpenMP runtime
- Is this computational pattern frequent? Can a modification of the OpenMP standard help handle this kind of pattern with less effort?

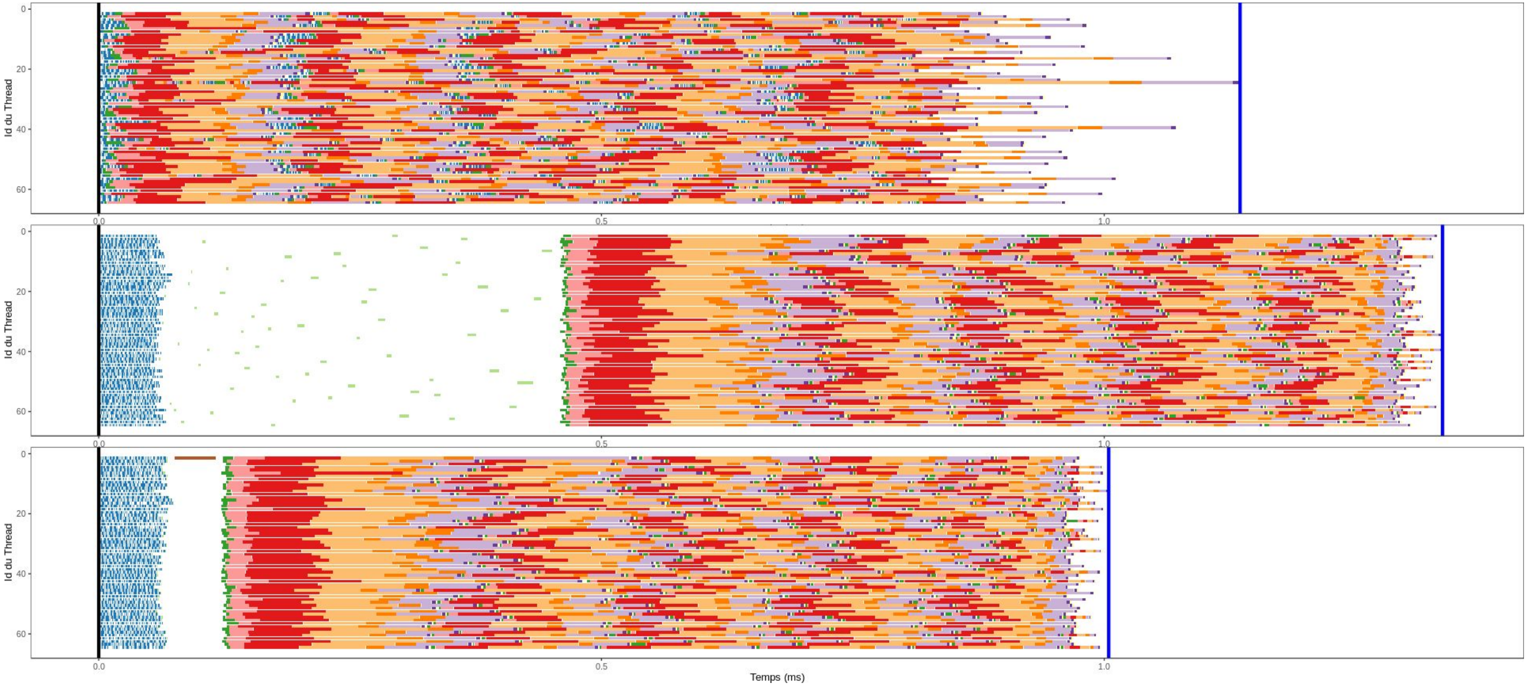
We have a dream...

```
#pragma omp for schedule(static)
for (auto i = 0; i < N; i++) {
  R = f_k ∘ ...f_2 ∘ f_1(indiv[i])
#pragma omp barrier schedule_modifier(\
    schedule(monotonic: dynamic, 1)\
    sort: R, [](r1, r2){return size(r1.dna) > size(r2.dna);})
  fitness[i] = f_n ∘ ...f_{k+2} ∘ f_{k+1}(R)
}
```

Thank you !

AEVOL: WORKFLOW OF A GENERATION

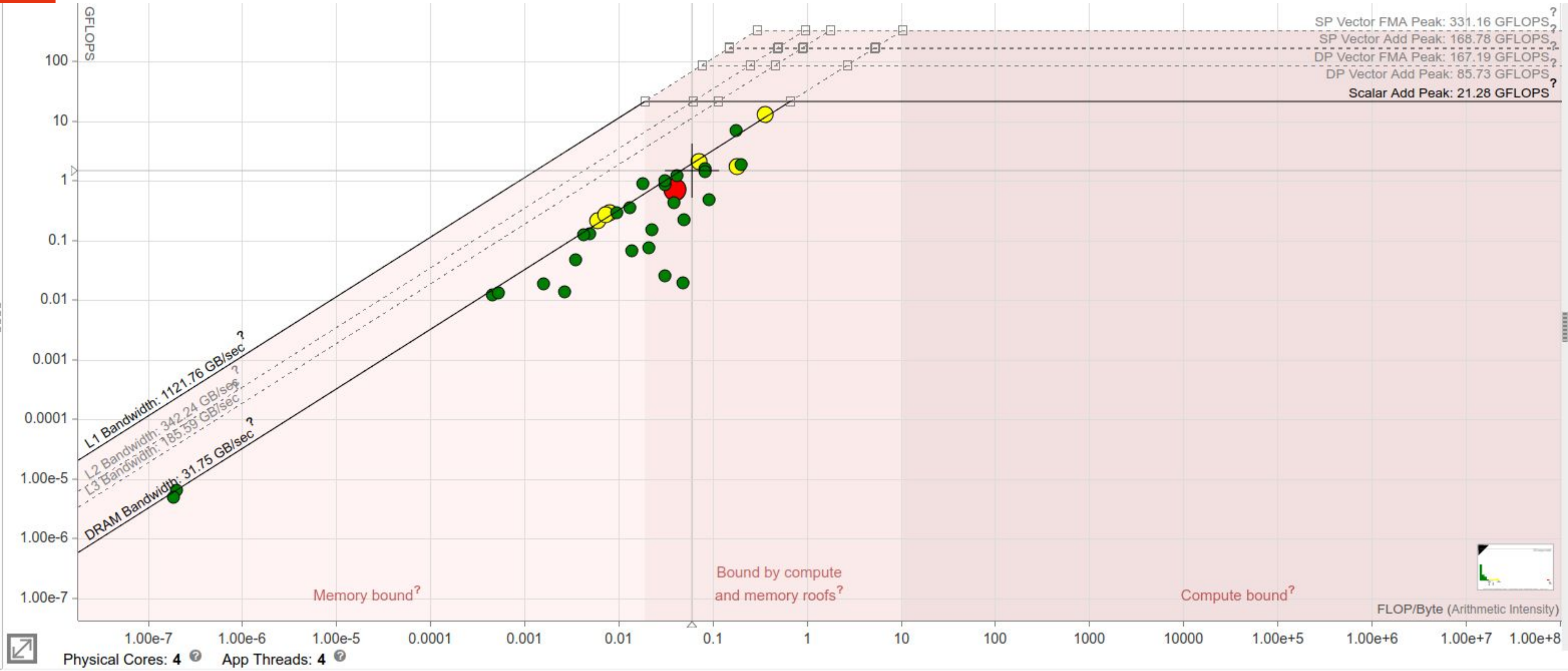




dynamic

OpenMP reduction

Our final Solution


GFLOPS
SP Vector FMA Peak: 331.16 GFLOPS
SP Vector Add Peak: 168.78 GFLOPS
DP Vector FMA Peak: 167.19 GFLOPS
DP Vector Add Peak: 85.73 GFLOPS
Scalar Add Peak: 21.28 GFLOPS
L1 Bandwidth: 1121.76 GB/sec
L2 Bandwidth: 342.24 GB/sec
L3 Bandwidth: 185.59 GB/sec
DRAM Bandwidth: 31.75 GB/sec
Memory bound
Bound by compute and memory roofs
Compute bound
FLOP/Byte (Arithmetic Intensity)
100
10
1
0.1
0.01
0.001
0.0001
1.00e-5
1.00e-6
1.00e-7
1.00e-7
1.00e-6
1.00e-5
0.0001
0.001
0.01
0.1
1
10
100
1000
10000
1.00e+5
1.00e+6
1.00e+7
1.00e+8
Physical Cores: 4
App Threads: 4