veryfastFVA: A hybrid MPI/OpenMP parallel implementation of flux variability analysis

Marouen Ben Guebila[1] and Ines Thiele[1*]

***Abstract*— Genome scale metabolic models (GSMMs) of living organisms are used in a wide variety of applications pertaining to health and bioengineering. They are formulated as linear programming (LP) problems that are often times under-determined. Flux Variability Analysis (FVA) characterizes the Alternative Optimal Solution (AOS) space enabling thereby the assessment of the solution's robustness. fastFVA (FFVA), the C implementation of MATLAB FVA, allowed to gain substantial speed up, although parallelism was managed through MATLAB. We present, veryfastFVA (VFFVA), a pure C implementation of FVA, that relies on a hybrid MPI/OpenMP management of parallelism. The flexibility of VFFVA allowed to gain between 2 and 20 speed-up factor and decreases memory usage 14 fold in comparison to fastFVA. Finally, VFFVA allows to process a higher number of GSMMs in faster times accelerating thereby biomedical simulation.**

I. INTRODUCTION

Modeling and simulation of biological systems gained tremendous interest thanks to the increasing predictive ability of modeled systems in healthcare and in the biotechnology industry . Microbial and human systems are most amenable to modeling given the wealth of data in the literature along with the development of computational methods.
Particularly, constraint-based reconstruction and analysis (COBRA) methods enable the reconstruction of the metabolism of biological systems *in silico* as a linear program (LP) [1]. Subsequently, an objective function of the system is formulated and optimized for e.g. biomass yield, metabolite production. Although the objective is uniquely determined, the set of corresponding solutions forms the space of alternative optimal solutions (AOS) that describes the possible conditions in which the optimal objective is achievable. AOS is quantified using flux variability analysis (FVA) [2], which provides a range of minimum and maximum values for each variable of the system. Biologically, these values overlap with the fitness of a given system to achieve optimality. This way, simulations can be easily cross validated with experimental values.
fastFVA (FFVA) [3], a recent implementation of FVA gained tremendously in speed over the `fluxvariability` COBRA toolbox MATLAB function [4]. Two main improvements were the driving factor of the gained efficiency: first, the C implementation of FVA which allowed a higher flexibility using the CPLEX C API in comparison to MATLAB. The second was the use of the same LP object in the different iterations, which avoided solving the problem from scratch, thereby saving presolve time. FFVA is compiled as MATLAB Executable (MEX) file, that can be called from MATLAB directly.
Although, given the exponentially growing size of the models, FFVA is ran in parallel in most cases. Parallelism simply relies on allocating the cores through MATLAB `parpool` function and running the iterations through `parfor` loop. The load is statically balanced over the workers such as they process an equal amount of iterations, which does not guarantee an equal processing time among the workers. Often times, workers process their chunk of iterations and stay idle, waiting to synchronize with the remaining workers, which can result in less efficient running times.
We present veryfastFVA (VFFVA), a pure C implementation of FVA, that has a lower level management of parallelism over fastFVA. The program is provided as a standalone, that does not rely on MATLAB thereby offering an open source alternative for constraint-based biological analysis. The major contribution lies in the management of parallelism through a hybrid OpenMP/MPI, for shared memory and non-shared memory systems respectively, which offers great flexibility and speed-up over the existing implementations. While keeping the up-mentioned advantages of fastFVA, load balancing in VFFVA was scheduled dynamically [5] in a way to guarantee equal running times between the workers. The input does not rely on MATLAB anymore as the LP problem is read in the industry standard $.mps$ file, that can be also obtained from the classical $.mat$ files through a provided converter. The improvements in the implementation allowed to gain from 1.6 to 20 speed-up factor and reduces memory requirements 14 fold in comparison to fastFVA and the *Julia*-based `distributedFBA` implementation [6], in a similar parallel setting.
Taken together, metabolic models are steadily growing in number and complexity and their analysis requires the design of efficient tools. VFFVA allows to make the most of modern machines specifications in order to run a greater amount of simulation in less time thereby enabling biological discovery.

II. MATERIAL AND METHODS

A. Flux variability analysis

The LP problem has dimension n, constrained by matrix $S_{(m,n)}$ and bounded by lower bound $lb_{(n,1)}$ and upper bound $ub_{(n,1)}$ vectors. An initial LP optimizes for the objective function of the system to obtain a unique optimum e.g. biomass maximization, like the following:

[1]Molecular Systems Physiology group at the Luxembourg Centre for Systems Biomedicine, University of Luxembourg, Campus Belval.
*corresponding author: `ines.thiele@uni.lu`

$$\begin{array}{ll} \text{maximize} & Z_1 = c^T v \\ \text{subject to} & \\ & S.v = b \\ & lb < v_i < ub \end{array} \tag{1}$$

The system being under-determined $(m < n)$, there can be an infinity of solution vectors $v_{(n,1)}$ that satisfy the unique optimal objective $(c^T v)$, with $c_{(n,1)}$ as the objective coefficient vector. In order to delineate the space of alternative optimal solutions (AOS), the objective function is set to its optimal value, while iterating over the n dimensions of the problem. Then each of the dimensions is set as a new objective function and is maximized and minimized for. The total number of LPs is then equal to $2n$. The problem is described as the following:

$$\begin{array}{lll} \text{iterate over} & [1, n] & \\ & \text{set} & c_i = 1 \\ & \text{max/min} & Z_i = c^T v \\ & \text{subject to} & \\ & & S.v = b \\ & & c^T v = Z_1 \\ & & lb < v_i < ub \end{array} \tag{2}$$

The obtained minimum and maximum objective value for each dimension defines the range of optimal solutions.

B. Management of parallelism

Problem 2 is entirely parallelizable through allocating the $2n$ LPs among the available workers. The strategy used so far in the existing implementations was to divide $2n$ equally among the workers. Although, the solution time can vary widely between each LP.
In shared memory systems, Open Multi-Processing (OpenMP) C API allows to balance the load among the threads dynamically such that every instruction runs for an equal amount of time. Since it is challenging to estimate *a priori* the running time of an LP, the load has to be adjusted dynamically, depending on the chunks of the problem processed by every thread.
In systems that do not share memory, Message Passing Interface (MPI) was used to create instances of Problem 2. Every process then calls the shared memory execution through OpenMP.
At the end, the final program is comprised of a hybrid MPI/OpenMP implementation of parallelism which allows a great flexibility of usage, particularly in High Performance Computing (HPC) setting.

C. Model description

A selection of models [3] was tested on FFVA and VFFVA. The models (Table I) are characterized by the dimensions of the stoichiometric matrix $S_{m,n}$. Each of them represent the metabolism of human and bacterial systems.

TABLE I: Model size and description

Model	Organism
Ecoli_core	*Escherischia coli*
Pputida	*Pseudomonas putida*
EcoliK12	*Escherischia coli*
Recon2	*Homo sapiens*
E_Matrix	*Escherischia coli*
E_c_Matrix	*Escherischia coli*
Harvey	*Homo sapiens*

Models pertaining to the same biological system with different S matrix size, have different levels of granularity and biological complexity.

III. Results

The OpenMP/MPI hybrid implementation of VFFVA allowed to gain important speed-up factors over the static load balancing in the MATLAB implementation. In this section, we benchmarked the running times of VFFVA in comparison to FFVA at different resource settings then we compared different strategies of load balancing and their impact on equalizing the running time per worker. While in fastFVA, the authors benchmarked serial runs, in the present work, the emphasis was placed upon parallel running times.

A. Parallel construct in a hybrid OpenMP/MPI setting

The MATLAB implementation of parallelism through the Parallel computing toolbox provides great ease-of-use, wherein two commands only are required to allocate and launch parallel jobs. Also, it saves the user the hassle of finding out if the jobs are run on memory sharing systems or not. VFFVA provides the user with a similar level of flexibility as it supports both types of systems. In addition, it allows to access advanced features of OpenMP and MPI such as dynamic load balancing. The algorithm starts first by assign chunks of iterations to every CPU (Fig1), in which a user defined number of threads simultaneously process the iterations. At the end, the CPUs synchronize and pass the result vector to the master CPU to reduce them to the final vector.
The main contributions of VFFVA are the complete use of C, which impacted mainly the computing time of small models $(n < 3000)$ and the dynamic load balancing that was the main speed-up factor for large models.

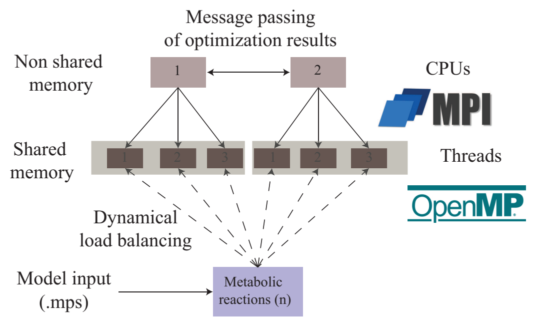

Fig. 1: hybrid OpenMP/MPI implementation of FVA.

B. Impact on computing small models

VFFVA and FFVA were run five times on each model. On small models, VFFVA had at least 20 fold speed-up (the differences between group averages are significant at $p < 0.05$) (Table II). The main contributing factor was the use of C over MATLAB in all steps of the analysis. In particular, the loading time of MATLAB java machine and the assignment of workers through `parpool` was much greater than the analysis time itself.
The result highlights the power of C in gaining computing speed, through managing the different low-level aspects of memory allocation and variable declaration.
In the analysis of large models, where MATLAB loading time becomes less significant, dynamic load balancing becomes the main driving factor of the gained speed-up.

TABLE II: Comparison of running times of FFVA and VFFVA in small models in seconds.

Model	FFVA mean(std) loading and analysis time	VFFVA mean(std) loading and analysis time	FFVA mean(std) analysis only
2 cores			
Ecoli_core	19.5(0.5)	0.2(0.01)	0.37(0.1)
Pputida	19.2(0.7)	0.6(0.02)	0.81(0.09)
EcoliK12	20.4(0.6)	2.2(0.06)	2.41(0.09)
4 cores			
Ecoli_core	19.6(0.6)	0.2(0.005)	0.32(0.01)
Pputida	19.4(1)	0.5(0.02)	0.61(0.01)
EcoliK12	20(0.8)	1.3(0.04)	1.64(0.08)
8 cores			
Ecoli_core	19.4(0.5)	0.2(0.03)	0.35(0.05)
Pputida	19.6(0.7)	0.4(0.04)	0.53(0.009)
EcoliK12	20(0.49)	0.9(0.01)	1.22(0.08)
16 cores			
Ecoli_core	20.2(0.4)	0.2(0.008)	0.41(0.05)
Pputida	19.5(0.4)	0.4(0.04)	0.51(0.03)
EcoliK12	22(0.7)	0.7(0.01)	0.87(0.03)
32 cores			
Ecoli_core	22.2(0.4)	0.3(0.008)	0.6(0.12)
Pputida	21.5(0.6)	0.4(0.01)	0.53(0.004)
EcoliK12	21.5(0.6)	0.6(0.03)	0.78(0.04)

C. Impact on computing large models

The speed-up gained on computing large models (Recon2 and ME) reaches three folds with VFFVA (Fig2). In fact, with dynamic load balancing enabled, VFFVA updates the assigned chunks of iterations to every worker. In this case, faster workers process more iterations, in such way that all workers synchronize at the same time to reduce the results. Particularly, VFFVA increased in speed as the model was bigger and the number of threads rose (Fig2-E_Matrix). We explored the different load balancing startegies (static, guided and dynamic) with our largest models (E_c_Matrix and Harvey).

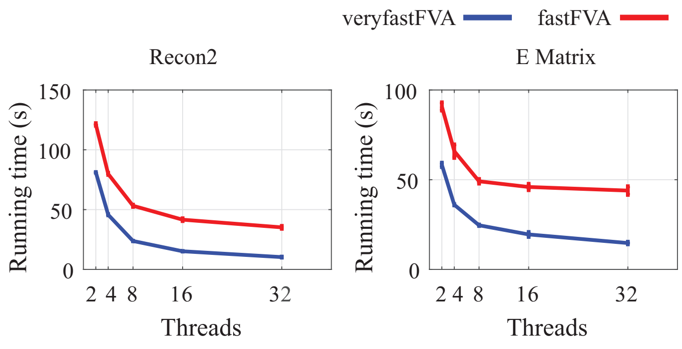

Fig. 2: Running times of Recon2 and E_Matrix model.

D. Load management

Load management describes the different approaches to assign iterations to the workers. It can be static, where an even number of iterations is assigned to each worker. Guided schedule refers to dividing the iterations in chunks of size $n/workers$ initially and $remaining\_iterations/workers$ afterwards. The difference with static lies in the dynamic assignment of chunks of the same size, in a way that fast workers can process more iteration blocks. The large scale model simulations (Fig2) used guided scheme. Finally, dynamic schedule is very similar to guided except that chunk size is given by the user, which allows a greater flexibility.

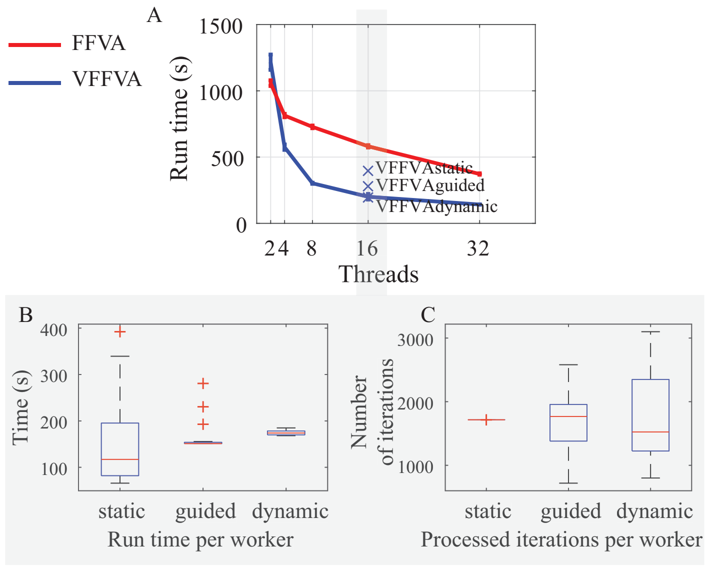

Fig. 3: Running times of E_c_Matrix model.

In the following section, we will compare the load balancing strategies of ME_c model at 16 threads.
1) Static schedule: Using static schedule, VFFVA assigned an equal number of iterations to every worker. With 16 threads, the number of iterations per worker equalled 1715 and 1716 (Fig3-C). Expectedly, the running time varied widely and resulted in a final time of $581s$.
2) Guided schedule: With guided schedule (Fig3-A), the highest speed-up (2.9) was achieved with 16 threads (Fig3-B). The running time per worker was quite comparable and

the iterations processed varied between 719 and 2581.

3) Dynamic schedule: Using a dynamic load balancing with a chunk size of 50 resulted in similar results to the guided schedule. The final running time equalled 197s, while FFVA took 581s. An optimal chunk size has to be small enough to ensure a frequent update on the workers load, and big enough to take advantage of the solution basis reuse in every worker. At a chunk size of 1 i.e. each worker is assigned 1 iteration at a time, the final solution time equalled 272s. The reason being that if the worker is updated quite often with new pieces of iterations, then it looses the stored solution basis of the previous problem and has to solve from scratch.

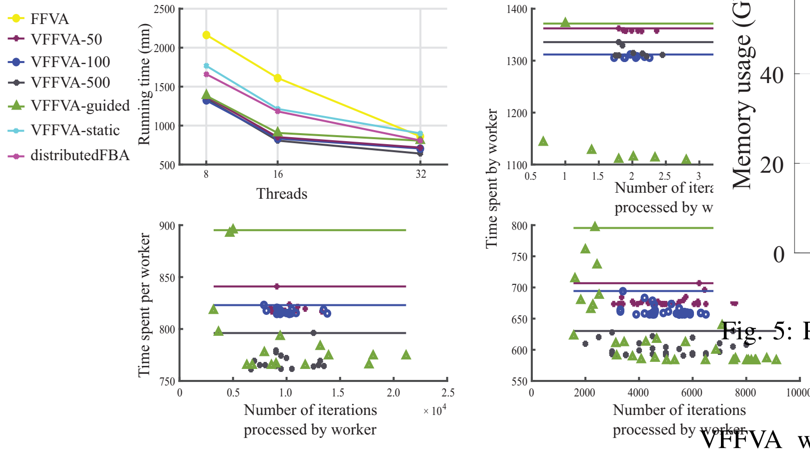

Fig. 4: Running times per worker of Harvey model.

Similarly, Harvey model (Fig 4-A) had a 2-fold speed-up with 16 threads using a chunk size of 50 (821 mn). Running times with guided schedule (1440 mn), dynamic schedule with chunk size 100 (847 mn) and chunk size 500 (1293 mn) were less efficient due to the slower update rate leading to a variable analysis time per worker (Fig 4-B,C,D). VFFVA on 8 threads (1306 mn with chunk size 50) proved comparable to FFVA (1611 mn) and distributedFBA (1247 mn) both on 16 threads, thereby saving computational resources and time.

E. Impact on memory usage

In MATLAB, the execution of n parallel jobs implies launching n instances of MATLAB. On average, one instance needs 2 Gb. In parallel setting, the memory requirements are at minimum $2n$ Gb, which can limit the execution of highly parallel jobs. In the *Julia*-based `distributedFBA`, the overall memory requirement exceeds 15 Gb at 32 cores. VFFVA requires only the memory necessary to load n instances of the input model. The differences between the FFVA and VFFVA get more pronounced as the number of threads increases (Fig5) i.e. 13.5 fold at 8 threads, 14.2 fold at 16 threads,14.7 fold at 32 threads.

Finally, VFFVA outran FFVA and distributedFBA both on execution time and memory requirements (Table III). The advantage becomes important with larger models and higher number of threads, which makes VFFVA particularly suited for analysing the exponentially-growing-in-size metabolic models in HPC setting.

TABLE III: Comparative summary of the methods' features.

Feature	VFFVA	distributedFBA	FFVA	FVA
Time	++++	+++	++	+
Memory	+++	++	+	+
Load balancing	dynamic	static	static	static

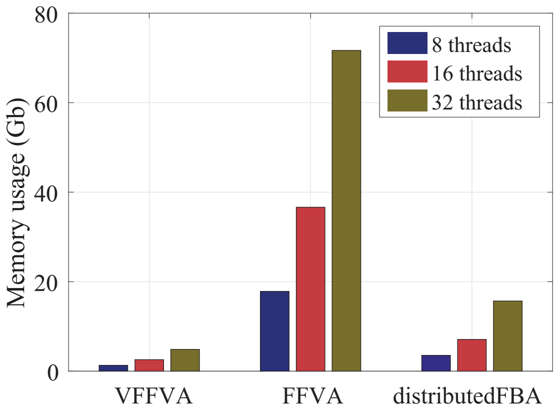

Fig. 5: Physical memory usage at 8, 16 and 32 threads.

IV. Hardware and Software

VFFVA was run on a Dell HPC machine with 72 Intel Xeon E5 2.3GHz cores and 768 GigaBytes of memory. The current implementation was tested with Open MPI v1.10.3, OpenMP 3.1, GCC 4.7.3 and IBM ILOG CPLEX free-of-charge academic version (12.6.3). FFVA was tested with MATLAB 2014b and distributedFBA was run on Julia v0.5. ILOG CPLEX was called with the following parameters:

```
PARALLELMODE=1
THREADS=1
AUXROOTTHREADS=2
```

Additionally, large scale coupled models with unscaling infeasibilites might require

```
SCAIND=-1
```

The call to VFFVA is done from bash as follows:

```
mpirun -np <nproc> --bind-to none -x
OMP_NUM_THREADS=<nthr> ./veryfastFVA
<model.mps> <scaling>
```

with *nproc* is the number of non-shared memory processes, *nthr* is the number of shared memory threads, *scaling* is CPLEX scaling parameter where 0 leaves it to the default (equilibration) and -1 sets it to unscaling. For large models, OpenMP threads were bound to physical cores through setting the environment variable

```
OMP_PROC_BIND=TRUE
```

while for small models, setting the variable to `FALSE` yielded faster running times. The schedule is set through the environment variable

```
OMP_SCHEDULE=<schedule,chunk>
```

where `schedule` can be `static`, `dynamic` or `guided`, and `chunk` is the minimal number of iterations processed per worker at a time. Benchamarking was done using $time$ function of bash.

ACKNOWLEDGMENT

The authors would like to acknowledge fastFVA authors for publicly sharing their code, IBM for providing a free academic version of ILOG CPLEX as well as Valentin Plugaru and Dr. Laurent Heirendt at the University of Luxembourg, for useful comments and guidance. The experiments presented in this paper were carried out in part using the HPC facilities of the University of Luxembourg [7] – see `http://hpc.uni.lu`.

REFERENCES

[1] E. J. OBrien, J. M. Monk, and B. O. Palsson, "Using genome-scale models to predict biological capabilities," *Cell*, vol. 161, no. 5, pp. 971–987, 2015.

[2] R. Mahadevan and C. Schilling, "The effects of alternate optimal solutions in constraint-based genome-scale metabolic models," *Metabolic engineering*, vol. 5, no. 4, pp. 264–276, 2003.

[3] S. Gudmundsson and I. Thiele, "Computationally efficient flux variability analysis," *BMC bioinformatics*, vol. 11, no. 1, p. 489, 2010.

[4] S. A. Becker, A. M. Feist, M. L. Mo, G. Hannum, B. Ø. Palsson, and M. J. Herrgard, "Quantitative prediction of cellular metabolism with constraint-based models: the cobra toolbox," *Nature protocols*, vol. 2, no. 3, pp. 727–738, 2007.

[5] M. Süß and C. Leopold, "Common mistakes in openmp and how to avoid them," in *OpenMP Shared Memory Parallel Programming*, pp. 312–323, Springer, 2008.

[6] L. Heirendt, R. M. Fleming, and I. Thiele, "Distributedfba. jl: High-level, high-performance flux balance analysis in julia," *arXiv preprint arXiv:1611.04743*, 2016.

[7] S. Varrette, P. Bouvry, H. Cartiaux, and F. Georgatos, "Management of an academic hpc cluster: The ul experience," in *Proc. of the 2014 Intl. Conf. on High Performance Computing & Simulation (HPCS 2014)*, (Bologna, Italy), pp. 959–967, IEEE, July 2014.