Testing of the MPI interface to the DragonFly parallel toolbox.

Version – 1.0

Revision history

Rev.	*Date*	*By*	*Revision summary*
1.0	09-09-2013	Ivano Azzini	First release

Table of Contents

1 Executive Summary

In this document we describe some tests/ experiments performed using DRAGONFLY package

2 INTRODUCTION

2.1 DELIVERABLE OBJECTIVES

Identify routines and algorithms can be subject to parallelization using DRAGONFLY package [1-5] and perform consistency checks of the results and speed-up tests..

2.2 PARALLEL TOOLKIT DEFINITIONS, ACRONYMS, ABBREVIATIONS

MONFISPOL: Monetary and Fiscal Policy FP7 Project.

FP7: Seventh Framework Programme.

DYNARE: Dynare is a software platform for handling a wide class of economic models, in particular dynamic stochastic general equilibrium (DSGE) and overlapping generation (OLG) models.

MPI: is a standardized and portable message-passing system to function on a wide variety of parallel computers

Matlab: is a high-level technical computing language and interactive environment for algorithm development, data visualization, data analysis, …

Octave: is a high-level interpreted language, primarily intended for numerical computations.

DRAGONFLY: routine's package name.

GUI: Graphical User Interface.

2.3 STRUCTURE OF THE DOCUMENT

Section 3 - GENERAL CONTEXT AND BAKGROUND.
Section 4 - DRAGONFLY TESTING
Section 5 - CONCLUSION

3 GENERAL CONTEXT AND BACKGROUD

The JRC, in the framework of the MONFISPOL FP7 project, has developed a parallel implementation of MATLAB/Octave, Windows/Unix that works under the DYNARE environment. This package has been designed in such an abstract and general form that allows to extend its application outside the DYNARE framework into any generic MATLAB/Octave, Windows/Unix computation. Nevertheless, the current implementation requires that the software developer manually extracts the loop to be run in parallel from a generic routine "floop.m" into a subroutine "flop_core.m" with some specified list of inputs and with explicitly wrapping and sending to slaves the list of local and global variables that are needed to run the loop in parallel. In other words, no MPI interface is available to link to the parallel package.

So we have analyzed and identified a set of algorithms (the DragonFly package) to build simple routines/keywords to be used in a generic routine "floop.m" such that the loops to be run in parallel are automatically extracted by the parallel package, without the need for software developers to explicitly break the progenitor routine into the group [floop.m, flop_core.m]. This will allow an easier and more widespread use of the free parallel implementation of MATLAB/Octave, Windows/Unix developed at JRC.

The DRAGONFLY approach to parallelism is very simple and easy to use, the portion of code (generally a 'for' cycle) that must be executed in parallel (local, using multi-core computer or remotely on a computer cluster), must be enclosed with two "keywords", i.e two MATLAB functions: **DRAGONFLY_Parallel_Start** and **DRAGONFLY_Parallel_Block_End.**
For example we can consider the following MATLAB code:

```
…
for i=1:n
   PnP=CheckIfIsPrime(i);
   if PnP==1
      Replay(rI)=i;
      rI=rI+1;
   end
end
…
```

This code computes all the prime numbers less than n, and it is executed in a traditional sequential way.

The following modified code is executed in parallel on the cores\computer cluster defined by the user, with the same final results.

```
…
startFor=1;
endFor=n;
[ … output … ]=DRAGONFLY_Parallel_Start( … variable … );
for i= startFor: endFor
   PnP=CheckIfIsPrime(i);
   if PnP==1
      Replay(rI)=i;
      rI=rI+1;
   end
end
DRAGONFLY_Parallel_Block_End(… variable …);
…
```

According to the criteria summarized in Section 3., we can see as the main goal of these two functions is to automatically execute the manual work required in parallel DYNARE, and have a paradigm to write parallel codes similar to MATLAB native mechanism.

Specifically, the **DRAGONFLY_Parallel_Start** has to perform the following tasks:

- Extract the MATLAB code written by the user until the next keyword: **DRAGONFLY_Parallel_Block_End.**
- Check if there are syntax errors: for example, if there is no command 'for', if there are too many keywords, and so on.
- Manage and retrieve all MATLAB variables (global, local and persistent) in order to have a parallel execution equivalent to its sequential execution.
- Perform a parsing of the input code that is able to reproduce the flop_core.m function as described in section 3. During the parsing step, the function **DRAGONFLY_Parallel_Start** can also take in account user options, like the visualization of computational state.
- Link to the masterParallel function (the parallel wrapper of Parallel DYNARE [1-5]) with in input the flop_core.m function and the hardware scenario define by the user.

- Wait and collect the output of masterParallel function. Delete the dynamic flop_core.m. Finally implement a mechanism to avoid a second serial execution of 'for' code after completion of the parallel session and 'send' the result to **DRAGONFLY_Parallel_Block_End** function.

The **DRAGONFLY_Parallel_Block_End** function takes as input the results of materParallel computation, and stacks them so that there are no differences between the parallel/sequential execution for the input code. For example if the serial code produces an output variable of dimension [100x1], a parallel session may produce four tokens of size [25x1]. The function **DRAGONFLY_Parallel_Block_End** will re-combine the parallel outputs to reproduce the serial ones.

The Dragonfly package is composed of about 40 MATLAB/Octave functions, which can be grouped as:

1. the couple **DRAGONFLY_Parallel_Block_Start** and **DRAGONFLY_Parallel_Block_End** functions,

2. hardware definition, analysis, and testing functions,

3. management and synchronization functions,

4. communication functions,

5. visualization functions,

6. utility functions.

As reported in 3, DRAGONFLY is derived from parallel DYNARE, therefore it makes use of many parallel DYNARE functions. The documentation for this set of MATLAB/Octave functions can be found in [1-5] and in many others similar papers/documents. Here, we describe in detail the functions and features added in DRAGONFLY package. The most important functions and features inserted in DRAGONFLY, are:

- the graphical user interface that can be used to define, analyze, and test the hardware environment, as alternative to the traditional MATLAB/Octave functions (some functions in group 2),

- the two functions **DRAGONFLY_Parallel_Block_Start** and **DRAGONFLY_Parallel_Block_End** functions (point 1.),

- the management of 'break' instances (if they exist) inside the 'for' loop to parallelize. This additional and useful feature has required to modify the functions belonging group 1, 3, 4, and 5.

To simplify the hardware definition, test, ... etc. that will be involved in the parallel computation, we have developed a specific GUI, (see Figure 1.) that can be used as alternative to the traditional MATLAB/Octave functions as described in [1-5].

Using the GUI in Figure 1, the user can easily define the computer cluster (the hardware setting box), fix all the other possible options, as described in [1-5] (the option box) and use some useful utility to visualize the computer cluster configuration, to reset the computational environment, ... etc. (the utility box). Finally with the "Test the cluster configuration" button the user can test the computer hardware, the network, ... etc.

The greatest innovation in DRAGONFLY, are the two functions **DRAGONFLY_Parallel_Block_Start** and **DRAGONFLY_Parallel_Block_End**. Following the most recent paradigm in parallel computing theory, these functions allow, to easy execute a generic 'for' loop in parallel on a computer clustering. The function **DRAGONFLY_Parallel_Block_Start** is built with the logic described above and is defined as:

```
function [startFor, endFor, foutEnlarged] = DRAGONFLY_Parallel_Start(loopVar, startFor, endFor, NamFileInput)
```

The inputs of this function are the loop var of the 'for' cycle to parallelize (loopVar), its initial value (startFor) and its final value (endFor). The input variable NamFileInput is optional, and it stores the list of file required by some machines in computer cluster in order to properly perform the parallel computation.

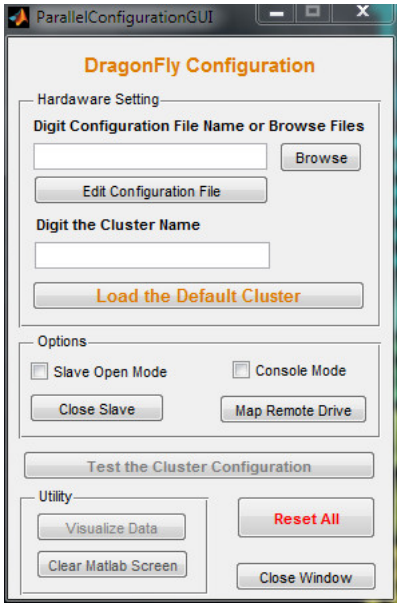

Figure 1. The DRAGONFLY hardware configuration.

The output of **DRAGONFLY_Parallel_Block_Start** (besides startFor and endFor) is the output of masterParallel function (`foutEnlarged`), with some additional information (nBlockPerCPU, and totCPU) [1-5].

The **DRAGONFLY_Parallel_Block_Start** function also implements a mechanism to manage user errors. It is, for example, able to verify if:

- the parallel keywords (**DRAGONFLY_Parallel_Block_ (Start, End))** don't match,
- there are two or more portions of code to parallelize (nested 'for' loop),
- remove redundant lines and spaces in a code.

...

On the other hand the function **DRAGONFLY_Parallel_Block_End** is defined as:

```
function DRAGONFLY_Parallel_Block_End(foutEnlarged,varargin);
```

The input var `foutEnlarged is the output of` **DRAGONFLY_Parallel_Block_Start.** 'Varargin' allows **DRAGONFLY_Parallel_Block_End** to accept any number of input arguments, namely a list of variables with the information necessary for reshaping and concatenating them properly. For example:

```
DRAGONFLY_Parallel_Block_End(foutEnlarged,'var1','2','var2','1','var3','1',...,'varN','2');
```

The var1 is concatenated vertically, var2 horizontality, and so on.

Finally we have implemented a mechanism to manage (if it exists) the key word 'break' within the 'for' loop that must be executed in parallel. In general, when a loop 'for' is performed with the aim to

find, if existing, a single object or a single condition (for example if there is a divisor of an integer n), when the object or condition is found, the cycle can be interrupted. To have an efficient computation, this must also be true when the 'for' loop is executed in parallel on a computer cluster. Then if a core or a machine in a cluster finds the required object or condition, it has to send a message to masterParallel function, then the masterParallel can send a message to the cores/machines involved in a cluster that the computation has to break.

4 DRAGONFLY TESTING

As described above, from a theoretical point of view we can use DRAGONFLY, to easily parallelize any 'for' loop used in a MATLAB/Octave functions/script. Nevertheless only 'for' cycles computationally expensive can be efficiently parallelized in order to reduce the total computational time of an algorithm. So to verify consistency check and speed-up in time we use DRAGONFLY to:

- check if an integer n is prime (the function `CheckIfIsPrime.m`),
  and
- to find all the prime numbers less than n (the function `FindAllPrimeNumbersLessThan.m`).

We use the "brutal" Approach to check if an integer n is prime or not: given n as input we simply verify if there exists a number in [2, n/2] that is a divisor of n.
This computational scenario allows us to test many capability of DRAGONFLY software:

- the results correctness,
- the absolute reduction in computational time, because the function `FindAllPrimeNumbersLessThan.m` is consists only of a 'for' cycle,
- we have a reduction in computational time only when n starts to become relatively big [1]. The total computational time for `CheckIfIsPrime.m` and `FindAllPrimeNumbersLessThan.m`, can vary between zero (n=2) and infinity (n very very big). So given the available hardware (number and type of computer, number of core, network speed...), these two functions can be used to fix when the parallel version becomes faster than the equivalent serial implementation and what is the relative reduction of computational time.

- the `FindAllPrimeNumbersLessThan.m` MATLAB routine calls the function `CheckIfIsPrime.m`, so we can use in `FindAllPrimeNumbersLessThan.m` the serial version of `CheckIfIsPrime.m`, to verify the correctness in simple and complex variable management. But also the parallelized version of `CheckIfIsPrime.m` to check the correctnesses DRAGONFLY in nested parallel function management,
- finally the function `FindAllPrimeNumbersLessThan.m` displaies their computational results in a monitor and save it on a structured file. This allows us to test the management of files on disk by DRAGONFLY.

These two functions are provided with the DRAGONFLY package, and the user can use them to perform preliminary tests for the a specific hardware scenario.

Following the above points, we use `CheckIfIsPrime.m` to verify if little integer numbers are prime, but also for relatively big n, for example:

12289
479001599
63018038201
...

In the same way we use the MATLAB function `FindAllPrimeNumbersLessThan.m`, to find the list of prime number less than n, for little n, but also for relatively large n.

In all the tests performed using these two functions the results were correct and saved correctly in a formatted text file.

For example if n=12289,

`CheckIfIsPrime(n)`, provide as results 'Yes', and

`FindAllPrimeNumbersLessThan(n)` provide as results the vector [1 2 3 5 ... 12269 12277 12281 12289]. This vector contain 1471 elements i.e. All the prime number nP<12289.

On the other hand, if n=12289-1=12288:

`CheckIfIsPrime(n)`, provide as results 'No, and

`FindAllPrimeNumbersLessThan(n)` provide as results the vector [1 2 3 5 ... 12269 12277 12281]. This vector contain 1470 elements i.e. All the prime number nP<12288.

Given a specific hardware, the reduction in computational time emerges only when n starts to become relatively large, which is in line with the results explained in [1]. We evaluate the computational time for `FindAllPrimeNumbersLessThan.m`, using n as input parameter on a laptop with a Celeron 847 dual core processor [6]. In parallel computing Speedup is defined by the following formula:

$$S_p = \frac{T_1}{T_P}$$

where:

- p is the number of processors

- T_1 is the execution time of the sequential algorithm

- T_p is the execution time of the parallel algorithm with *p* processors

Linear speedup or ideal speedup is obtained when $S_p = T_P$. So in our case with only two processors, deal speedup is 2 [7].

In Table 1, are reported some results.

n	Sequential Computational Time (sec)	Parallel Computational Time (sec)
100	0.03	4
1000	0.06	4.2
10.000	0.6	5.2
100.000	21	20
200.000	77	59
250.000	117	95
300.000	164	133
400.000	310	240
500.000	439	340
750.000	899	734
1.000.000	1582	1260

Table 1. The Computational Time.

We plot the value in Table 1, in Figure 2.

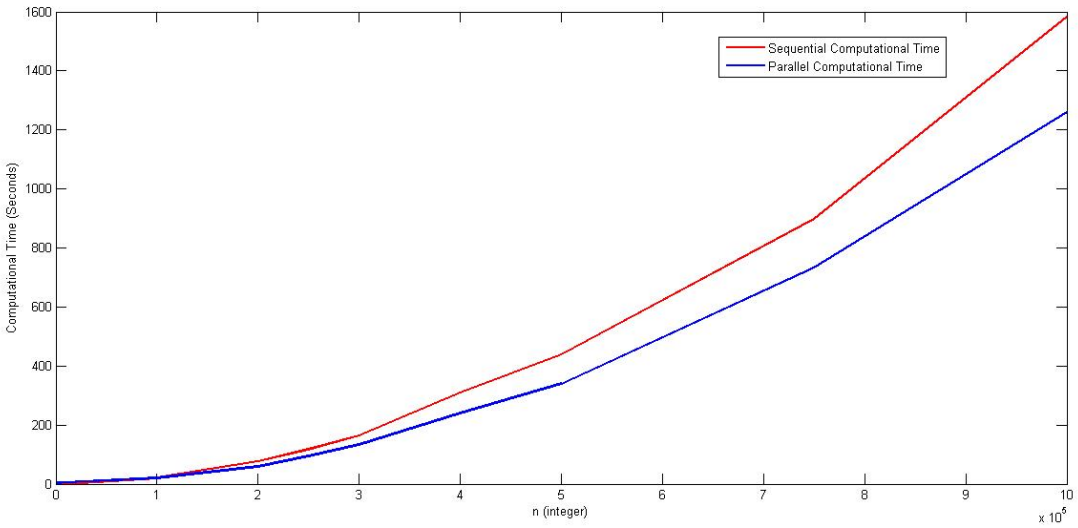

Figure 2 . The Computational Time.

From Figure 2, we can see the progressive reduction in computational time. For n=1.000.000, we an speed up of 1.25, and we expect that the speed increases even as n increases. Nevertheless, note as in this computational setting the optimal speed up can not be reached, because the prime number distribution is not uniform in N.

5 Conclusion

At the present stage we have a first full implementation of DRAGONFLY package that can be effectively used for running MATLAB/Octave codes in parallel. Furthermore the above tests show as DRAGONFLY, provides always correct results and greatly reduces the computation time!

REFERENCES

[1] Azzini I., Girardi R. and Ratto M., Parallelization of Matlab codes under Windows platform for Bayesian estimation: A Dynare application, DYNARE CONFERENCE September 10-11, 2007 Paris School of Economics.

[2] Azzini I., Ratto M., Parallel DYNARE Toolbox, 17th International Conference on Computing in Economics and Finance (CEF 2011). Society for Computational Economics Sponsored by the Federal Reserve Bank of San Francisco, June 29 through July 1, 2011.

[3] Ratto, M., Report on alternative algorithms efficiency on different hardware specifications and Alpha-version of parallel routines, (2010) MONFISPOL Grant no.: 225149, Deliverable 2.2.1, March 31, 2010.

[4] Ratto, M., Azzini, I., Bastani, H., Villemot, S., Beta-version of parallel routines: user manual, (2011) MONFISPOL Grant no.: 225149, Deliverable 2.2.2, July 8, 2011.

[5] **http://www.dynare.org/DynareWiki/ParallelDynare.**

[6] www.**intel**.com.

[7] John Benzi; M. Damodaran (2007). "Parallel Three Dimensional Direct Simulation Monte Carlo for Simulating Micro Flows". *Parallel Computational Fluid Dynamics 2007: Implementations and Experiences on Large Scale and Grid Computing*. Parallel Computational Fluid Dynamics. Springer. p. 95. Retrieved 2013-03-21.