The DRAGONFLY Package

Index

1. Introdunction

This document is very simple manual for the users of DRAGONFLY parallel software. DRAGONFLY package is a generalization (domain independent) of parallel DYNARE, and it is derived directly from parallel DYNARE. For this reason, the users can find others useful documentation in:

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.

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.

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.

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.

And http://www.dynare.org/DynareWiki/ParallelDynare,

Although the use DRAGONFLY is much simpler and more intuitive than using parallel DYNARE for the presence of a GUI, **we recommend you to read these documentations before using DRAGONFLY.**

2. The contents of DRAGONFLY package.

The DRAGONFLY package consists of:

- a directory called 'parallel' and containing a set of functions that are the DRAGONFLY core,
- a directory called 'DocumentationAndExamples' that contains this document and some example files.

DRAGONFLY is a free software then user can modify the core functions in accord with their necessity.

3. Some Parallel Programs concepts.

Parallel DYNARE is devoted to parallelize only DYNARE routines, DRAGONFLY modify parallel DYNARE matlab/octave functions and allow us to parallelize a generic matlab function. To show how this can be done, we use as example, some very basic questions in number theory: we simply use many different and naive schemes to find all the divisors of a integer n in order to derive certain property of n. For example the divisors of n say us if n is prime, a factorization for n, etc. Remember that: a PRIME number is a natural number (n) greater than 1 that has no positive divisors other than 1 and itself.
A brutal, "stupid" and serial strategy to determine all the divisors of n is:

```
compute n/2,
for any integer i in 2 to n/2,
compute n/i=c,
if n=i*c, save i in array D (i is a divisors for n)
returns D.
```

From D we can deduce some property of n, for example if D contains only 1 and n, then n is prime. So if we are only interested in determining whether a number is prime, we can rewrite the code above in this way:

```
...
if i is a divisor for n, stop and we can say: n
```

```
is not prime!
Othewise p is prime.
```

These two algorithms can be easily rewritten in parallel - 'thinking in parallel':
if we have p processors, split the set (n/2-2) in p subset: (n/2-2)/p and assign this portion of integers at any processor.
Any processor p can, independently from the others, to perform the necessary divisions in order to find all the divisors of n, or the first divisors (if exist), or the big divisor, or, etc.
Given this scenario, we have a lot of implementation details covering many parallel issues:

1. If we look for all the divisors of n:
    1.1 The cores involved in parallel computing save the divisors of n once at the end of parallel job
    1.2 Each new divisors of n is saved just when it is found.
    1.3 ...

2. If we want to check if n is prime:
    2.1 the processors are a kind of committee: the p processors perform at the same time and independently the divisions. If a processor find a divisor for n, wait the other p-1 processor and finally the processors committee sentence: n is not prime.
    2.2 When a processor find a divisor for n, using the master parallel function send a message to the others p-1 processors: stop the work: p is not prime!

If we do not list the entire divisors set for n, the strategy 2.2 is very better!

    2.3 in this case we can also test the load assigned to each core, in fact when n is no a prime number some cores need to do more work than the other. This is related with the distribution of divisors in the (n/2-2)/p sub-intervals.
    2.4 ...

We can see how, this simple problem of finding the divisors of an integer n, covers many of the problems we might encounter when we try to efficiently parallelize a generic code.

4. Examples

4.1 Sequential Code

Now we give a serial algorithm able to find the divisors for the integer n and then rewrite it in a two different forms suitable to be executed in parallel using DRAGONFLY.

Serial Solution without saving (if exist) divisors.

```
INPUTS
      o   n [integer]   The number for which some people ask us: "It is
      Prime?" or "What about all the divisors of n?"
OUTPUT
      o   Reply [strct] A struct with two field: a string 'Yes/No', an
      array contained (if exist) the divisors for n!

Replay.isPrime='';
Replay.Divisors=0;

m=floor(n/2);
for i=2:enneM
     d=n/i;
    if d==ceil(d)
         Reply.isPrime='No';
         return;
     end
   end
 Reply.isPrime='Yes';
```

Now rewrite it in parallel!

4.2 The equivalent parallel implementations.

The two solutions below cover many parallel issues, as described in section 3. The users can find others numerous and (we hope) useful information in (`nIsPrime.m` `nIsPrime_core.m`) and (`findAllDivisors.m` `findAllDivisors_core.m`) matlab/octave functions comment ('%'). In addition to these matlab functions, users can also find a cluster definition file (.txt) as example.

Example 1: The function : `nIsPrime.m and nIsPrime_core.m`

In this case, in accord with the strategy implemented in a distributeJobs.m Dragonfly native function, the available cores in a cluster, the user's weights, etc. we split the set [2 n/2] in some subsets we call them nC. After for these subsets each core separately try to find (if exist) the first divisor for n. We do not save the divisors for n, but simply find (if exist) the first divisor for n in the subsets of [2 n/2].
We note as, when n is no a prime number some core must do much more work than other cores. We think that this is related with the distribution of divisors in the nC above sub-interval and with the strategy used from us to find the first of them. The couple serial/parallel implementation for the solution are implemented in

`nIsPrime.m`
`nIsPrime_core.m`

Using these two functions, and the comments write within them, the user can learn how to parallelize a generic function using DRAGONFLY. Furthermore, when a core find a divisor for n, we use a mechanism to communicate to the other cores this fact, and then stop the computation! So in `nIsPrime_core` function we also implement a simple communication scheme between cores.

Example 2: The function : `findAllDivisors.m findAllDivisors_core.m`

In this case we want find and save all the divisors for n. This solution is very similar to example 1. The very big difference is the large amount of data that can be moved within the cluster. Some DRAGONFLY native functions are devoted to perform in an optimally way this task. Nevertheless, from user's point of view, the question is: for this cluster, it is better to save all the divisors of n one time at the end of job or save them and move them as soon as found by core? Users can modify the DRAGONFLY function to find the best solution for them. We think that a hybrid scheme is the best possible solution: as function of n we save the divisors of n into packets of size g(n) and then move them across the network (if necessary).

4.3 'Nested' usage

Users may need to call parallel functions within other functions repeatedly. In the functions `FindAllPrimeNumbersLessThan.m`, `nIsPrimeNested.m` and `nIsPrimeNested_core.m`, we show how this can be done efficiently.

5. Usage

Following the above examples (or the information in references),

1. users rewrite the code that wants to run in parallel,
2. define a cluster,
3. use a graphical interface or command line (for octave only command line) to define and then use cluster.

The use of graphical interface is very simple and intuitive, to use it run the function: ParallelConfigurationGUI.m. In the other side from command line (the function ParallelConfiguration.m) users we can provide zero to five inputs (four for octave because the variable 'Console Mode' is always set to 1) as described below:

0: the default environment is active i.e.: the first cluster in the file ([getenv('APPDATA'),'\dynare.ini']), with 'leave Slave Open' and 'Console Mode' deactivated, and no test is performed on the hardware!

1 to **5 (4)**: we search for a set of keywords as:

- The path and the name of the text file contained the cluster definition.

- 'Console'
  …
  'Console Mode'

- 'Slave'
  'Open'
  …
  'leave Slave Open'

- 'Test'
  …
  'Testing Hardware'

If (one or more) of this keyword are found the correspondent variable value is set to the opposite of default value. The remaining inputs (if exist) are interpreted as:
The **first**: the text file contained the cluster definition,
The **second**: the cluster's name in the above file. If the '**second**' does not exist is taken first cluster name in the list as the default setting.
As special case, the function ParallelConfiguration.m can be used also with only the keyword 'Close Slave' or 'CloseSlave', or 'closeSlave', … as input. In this case Dragonfly close, if exist, the matlab/octave slave open. In the other side, this operation is always performed when matlab is closed and user has used at least once the function ParallelConfiguration.m or ParallelConfigurationGUI.m. To do this we use the finish.m function placed in the shadow directory: .\DRAGOFLY\parallel\Exit from Parallel. The users can edit and modify this function.