Parallel DYNARE Toolbox
FP7 Funded
Project MONFISPOL Grant no.: 225149

Marco Ratto
European Commission, Joint Research Centre, Ispra, ITALY

October 1, 2026

Contents

1 The ideas implemented in Parallel DYNARE **3**

2 The DYNARE environment **4**

3 Installation and utilization **6**
3.1 Requirements . . . 7
3.1.1 For a Windows grid . . . 7
3.1.2 For a UNIX grid . . . 7
3.2 The user perspective . . . 7
3.2.1 The interface . . . 7
3.2.2 Preprocessing cluster settings . . . 10
3.2.3 Example syntax for Windows and Unix, for local parallel runs (assuming quad-core) 11
3.2.4 Examples of Windows syntax for remote runs . . . 12
3.2.5 Example Unix syntax for remote runs . . . 14
3.2.6 Testing the cluster . . . 15
3.3 The Developers perspective . . . 16
3.3.1 Write a parallel code: an example . . . 21

4 Parallel DYNARE: testing **25**
4.1 Test 1. . . . 25
4.2 Test 2. . . . 27
4.3 Test 3 . . . 29
4.4 Test 4 . . . 32

5 Conclusions **33**

A A tale on parallel computing **40**

Abstract

In this document, we describe the basic ideas and the methodology identified to realize the parallel package within the DYNARE project (called the "Parallel DYNARE" hereafter) and its algorithmic performance. The parallel methodology has been developed taking into account two different perspectives: the "User perspective" and the "Developers perspective". The fundamental requirement of the "User perspective" is to allow DYNARE users to use the parallel routines easily, quickly and appropriately. Under the "Developers perspective", on the other hand, we need to build a core of parallelizing routines that are sufficiently abstract and modular to allow DYNARE software developers to use them easily as a sort of 'parallel paradigm', for application to any DYNARE routine or portion of code containing computational intensive loops suitable for parallelization. We will finally show tests showing the effectiveness of the parallel implementation.

1 The ideas implemented in Parallel DYNARE

The basic idea behind "Parallel Dynare" is to build a framework to parallelize portions of code that require a minimal (i.e. start-end communication) or no communications between different processes, denoted in the literature as "embarrassingly parallel" (Goffe and Creel, 2008; Barney, 2009). In more complicated cases there are different and more sophisticated solutions to write (or re-write) parallel codes using, for example, OpenMP or MPI. Within DYNARE, we can find many portions of code with the above features: loops of computational sequences with no interdependency that are coded sequentially. Clearly, this does not make optimal use of computers having 2-4-8, or more cores or CPUs. The basic idea is to assign the different and independent computational sequences to different cores, CPU's or computers and coordinating this new distributed computational environment with the following criteria:

- provide the necessary input data to any sequence, possibly including results obtained from previous DYNARE sessions (e.g. a first batch of Metropolis iterations);
- distribute the workload, automatically balancing between the computational resources;
- collect the output data;
- ensure the coherence of the results with the original sequential execution.

Generally, during a program execution, the largest computational time is spent to execute nested cycles. For simplicity and without loss in generality we can consider here only `for` cycles (it is possible to demonstrate that any `while` cycle admits an equivalent `for` cycle). Then, after identifying the most computationally expensive `for` cycles, we can split their execution (i.e. the number or iterations) between different cores, CPUs, computers. For example, consider the following simple MATLAB piece of code:

```
...
n=2;
m=10^6;
Matrix= zeros(n,m);
for i=1:n,
    Matrix(i,:)=rand(1,m);
end,
Mse= Matrix;
...
```
Example 1

With one CPU this cycle is executed in sequence: first for `i=1`, and then for `i=2`. Nevertheless, these 2 iterations are completely independent. Then, from a theoretical point of view, if we have two CPUs (cores) we can rewrite the above code as:

```
            ...
            n=2;
            m=10^6;
            <provide to CPU1 and CPU2 input data m>

<Execute on CPU1>            <Execute on CPU2>
Matrix1 = zeros(1,m);         Matrix2 = zeros(1,m);
Matrix1(1,:)=rand(1,m);       Matrix2(1,:)=rand(1,m);
save Matrix1                  save Matrix2

            retrieve Matrix1 and Matrix2
            Mpe(1,:) = Matrix1;
            Mpe(2,:) = Matrix2;
```
Example 2

The `for` cycle has disappeared and it has been split into two separated sequences that can be executed in parallel on two CPUs. We have the same result (`Mpa=Mse`) but the computational time can be reduced up to 50%.

2 The DYNARE environment

We have considered the following DYNARE components suitable to be parallelized using the above strategy:

1. the Random Walk- (and the analogous Independent-)-Metropolis-Hastings algorithm with multiple chains: the different chains are completely independent and do not require any communication between them, so it can be executed on different cores/CPUs/Computer Network easily;

2. a number of procedures performed after the completion of Metropolis, that use the posterior MC sample:

   (a) the diagnostic tests for the convergence of the Markov Chain (`mcmc_diagnostics.m`);

   (b) the function that computes posterior IRF's (`posteriorIRF.m`).

   (c) the function that computes posterior statistics for filtered and smoothed variables, forecasts, smoothed shocks, etc.. (`prior_posterior_statistics.m`).

   (d) the utility function that loads matrices of results and produces plots for posterior statistics (`pm3.m`).

Unfortunately, MATLAB does not provide commands to simply write parallel code as in Example 2 (i.e. the pseudo-commands : `<provide inputs>`, `<execute on CPU>` and `<retrieve>`). In other words, MATLAB does not allow concurrent programming: it does not support multi-threads, without the use (and purchase) of MATLAB Distributed Computing Toolbox. Then, to obtain the behavior described in Example 2, we had to find an alternative solution.

The solution that we have found can be synthesized as follows:

When the execution of the code should start in parallel (as in Example 2), instead of running it inside the active MATLAB session, the following steps are performed:

1. *the control of the execution is passed to the operating system (Windows/Linux) that allows for multi-threading;*
2. *concurrent threads (i.e. MATLAB instances) are launched on different processors/cores/machines;*
3. *when the parallel computations are concluded the control is given back to the original MATLAB session that collects the result from all parallel*

'agents' involved and coherently continue along the sequential computation.

Three core functions have been developed implementing this behavior, namely `MasterParallel.m`, `slaveParallel.m` and `fParallel.m`. The first function (`MasterParallel.m`) operates at the level of the 'master' (original) thread and acts as a wrapper of the portion of code to be distributed in parallel, distributes the tasks and collects the results from the parallel computation. The other functions (`slaveParallel.m` and `fParallel.m`) operate at the level of each individual 'slave' thread and collect the jobs distributed by the 'master', execute them and make the final results available to the master. The two different implementations of slave operation comes from the fact that, in a single DYNARE session, there may be a number parallelized sessions that are launched by the master thread. Therefore, those two routines reflect two different versions of the parallel package:

1. the 'slave' MATLAB sessions are closed after completion of each single job, and new instances are called for any subsequent parallelized task (`fParallel.m`);

2. once opened, the 'slave' MATLAB sessions are kept open during the DYNARE session, waiting for the jobs to be executed, and are only closed upon completion of the DYNARE session on the 'master' (`slaveParallel.m`).

We will see that none of the two options is superior to the other, depending on the model size.

3 Installation and utilization

Here we describe how to run parallel sessions in DYNARE and, for the developers community, how to apply the package to parallelize any suitable piece of code that may be deemed necessary.

3.1 Requirements

3.1.1 For a Windows grid

1. a standard Windows network (SMB) must be in place;
2. PsTools (Russinovich, 2009) must be installed in the path of the master Windows machine;
3. the Windows user on the master machine has to be user of any other slave machine in the cluster, and that user will be used for the remote computations.

3.1.2 For a UNIX grid

1. SSH must be installed on the master and on the slave machines;
2. the UNIX user on the master machine has to be user of any other slave machine in the cluster, and that user will be used for the remote computations;
3. SSH keys must be installed so that the SSH connection from the master to the slaves can be done without passwords, or using an SSH agent.

3.2 The user perspective

We assume here that the reader has some familiarity with DYNARE and its use. For the DYNARE users, the parallel routines are fully integrated and hidden inside the DYNARE environment.

3.2.1 The interface

The general idea is to put all the configuration of the cluster in a config file different from the MOD file, and to trigger the parallel computation with option(s) on the `dynare` command line. The configuration file is designed as follows:

- be in a standard location
  - `$HOME/.dynare` under Unix;

- `c:\Documents and Setting\<username>\Application Data\dynare.ini` on Windows;

- have provisions for other Dynare configuration parameters unrelated to parallel computation
- allow to specify several clusters, each one associated with a nickname;
- For each cluster, specify a list of slaves with a list of options for each slave [if not explicitly specified by the configuration file, the preprocessor sets the options to default];

The list of slave options includes:

Name : name of the node;

CPUnbr : this is the number of CPU's to be used on that computer; if `CPUnbr` is a vector of integers, the syntax is `[s:d]`, with `d>=s` (`d, s` are integer); the first core has number 1 so that, on a quad-core, use `4` to use all cores, but use `[3:4]` to specify just the last two cores (this is particularly relevant for Windows where it is possible to assign jobs to specific processors);

ComputerName : Computer name on the network or IP address; use the NETBIOS name under Windows[1], or the DNS name under Unix.;

UserName : required for remote login; in order to assure proper communications between the master and the slave threads, it must be the same user name actually logged on the 'master' machine. On a Windows network, this is in the form `DOMAIN\username`, like `DEPT\JohnSmith`, i.e. user JohnSmith in windows group DEPT;

Password : required for remote login (only under Windows): it is the user password on `DOMAIN` and `ComputerName`;

RemoteDrive : Drive to be used on remote computer (only for Windows, for example the drive `C` or drive `D`);

[1] In Windows XP it is possible find this name in 'My Computer' − > mouse right click − > 'Property' − > 'Computer Name'.

RemoteDirectory : Directory to be used on remote computer, the parallel toolbox will create a new empty temporary subfolder which will act as remote working directory;

DynarePath : path to matlab directory within the Dynare installation directory;

MatlabOctavePath : path to MATLAB or Octave executable;

SingleCompThread : disable MATLAB's native multithreading;

Those options have the following specifications:

Node Options	type	default	Win		Unix	
			Local	Remote	Local	Remote
Name	string	(stop)	*	*	*	*
CPUnbr	integer or array	(stop)	*	*	*	*
ComputerName	string	(stop)		*		*
UserName	string	empty		*		*
Password	string	empty		*		
RemoteDrive	string	empty		*		
RemoteDirectory	string	empty		*		*
DynarePath	string	empty				
MatlabOctavePath	string	empty				
SingleCompThread	boolean	true				

The cluster options are as follows

Cluster Options	type	default	Meaning	Required
Name	string	empty	name of the node	*
Members	string	empty	list of members in this cluster	*

The syntax of the configuration file will take the following form (the order in which the clusters and nodes are listed is not significant):

```
[cluster]
Name = c1
Members = n1 n2 n3

[cluster]
Name = c2
Members = n2 n3

[node]
Name = n1
ComputerName = localhost
CPUnbr = 1

[node]
Name = n2
ComputerName = karaba.cepremap.org
CPUnbr = 5
UserName = houtanb
RemoteDirectory = /home/houtanb/Remote
DynarePath = /home/houtanb/dynare/matlab
MatlabOctavePath = matlab

[node]
Name = n3
ComputerName = hal.cepremap.ens.fr
CPUnbr = 3
UserName = houtanb
RemoteDirectory = /home/houtanb/Remote
DynarePath = /home/houtanb/dynare/matlab
MatlabOctavePath = matlab
```

Finally, the DYNARE command line options are:

- `conffile=<path>`: specify the location of the configuration file if it is not standard
- `parallel`: trigger the parallel computation using the first cluster specified in config file
- `parallel=<clustername>`: trigger the parallel computation, using the given cluster
- `parallel_slave_open_mode`: use the leaveSlaveOpen mode in the cluster
- `parallel_test`: just test the cluster, don't actually run the MOD file

3.2.2 Preprocessing cluster settings

The DYNARE pre-processor treats user-defined configurations by filling a new sub-structure in the `options_` structure, named `parallel`, with the following fields:

```
options_.parallel=
    struct('Local', Value,
    'ComputerName', Value,
    'CPUnbr', Value,
    'UserName', Value,
    'Password', Value,
    'RemoteDrive', Value,
    'RemoteFolder', Value,
    'MatlabOctavePath', Value,
    'DynarePath', Value);
```

All these fields correspond to the slave options except `Local`, which is set by the pre-processor according to the value of `ComputerName`:

Local: the variable `Local` is binary, so it can have only two values 0 and 1. If `ComputerName` is set to `localhost`, the preprocessor sets `Local = 1` and the parallel computation is executed on the local machine, i.e. on the same computer (and working directory) where the DYNARE project is placed. For any other value for `ComputerName`, we will have `Local = 0`;

In addition to the `parallel` structure, which can be in a vector form, to allow specific entries for each slave machine in the cluster, there is another `options_` field, called `parallel_info`, which stores all options that are common to all cluster. Namely, according to the `parallel_slave_open_mode` in the command line, the `leaveSlaveOpen` field takes values:

`leaveSlaveOpen=1` : with `parallel_slave_open_mode`, i.e. the slaves operate 'Always-Open'.

`leaveSlaveOpen=0` : without `parallel_slave_open_mode`, i.e. the slaves operate 'Open-Close';

3.2.3 Example syntax for Windows and Unix, for local parallel runs (assuming quad-core)

In this case, the only slave options are `ComputerName` and `CPUnbr`.

```
[cluster]
Name = local
Members = n1

[node]
Name = n1
ComputerName = localhost
CPUnbr = 4
```

3.2.4 Examples of Windows syntax for remote runs

- the Windows `Password` has to be typed explicitly;
- `RemoteDrive` has to be typed explicitly;
- for `UserName`, ALSO the group has to be specified, like `DEPT\JohnSmith`, i.e. user `JohnSmith` in windows group `DEPT`;
- `ComputerName` is the name of the computer in the windows network, i.e. the output of hostname, or the full IP address.

Example 1 Parallel codes that are run on a remote computer named `vonNeumann` with eight cores, using only the cores 4,5,6, working on the drive 'C' and folder '`dynare_calcs\Remote`'. The computer `vonNeumann` is in a net domain of the CompuTown university, with user `John` logged with the password `*****`:

```
[cluster]
Name = vonNeumann
Members = n2

[node]
Name = n2
ComputerName = vonNeumann
CPUnbr = [4:6]
UserName = COMPUTOWN\John
Password = *****
RemoteDrive = C
RemoteDirectory = dynare_calcs\Remote
DynarePath = c:\dynare\matlab
MatlabOctavePath = matlab
```

Example 2 We can build a cluster, combining local and remote runs. For example the following configuration file includes the two previous configurations but also gives

the possibility (with cluster name `c2`) to build a grid with a total number of 7 CPU's :

```
[cluster]
Name = local
Members = n1

[cluster]
Name = vonNeumann
Members = n2

[cluster]
Name = c2
Members = n1 n2

[node]
Name = n1
ComputerName = localhost
CPUnbr = 4

[node]
Name = n2
ComputerName = vonNeumann
CPUnbr = [4:6]
UserName = COMPUTOWN\John
Password = *****
RemoteDrive = C
RemoteDirectory = dynare_calcs\Remote
DynarePath = c:\dynare\matlab
MatlabOctavePath = matlab
```

Example 3 We can build a cluster, combining many remote machines. For example the following commands build a grid of four machines with a total number of 14 CPU's:

```
[cluster]
Name = c4
Members = n1 n2 n3 n4

[node]
Name = n1
ComputerName = vonNeumann1
CPUnbr = 4
UserName = COMPUTOWN\John
Password = *****
RemoteDrive = C
RemoteDirectory = dynare_calcs\Remote
DynarePath = c:\dynare\matlab
MatlabOctavePath = matlab

[node]
Name = n2
ComputerName = vonNeumann2
CPUnbr = 4
UserName = COMPUTOWN\John
Password = *****
RemoteDrive = C
RemoteDirectory = dynare_calcs\Remote
DynarePath = c:\dynare\matlab
MatlabOctavePath = matlab

[node]
Name = n3
ComputerName = vonNeumann3
CPUnbr = 2
UserName = COMPUTOWN\John
Password = *****
RemoteDrive = D
RemoteDirectory = dynare_calcs\Remote
DynarePath = c:\dynare\matlab
MatlabOctavePath = matlab

[node]
Name = n4
ComputerName = vonNeumann4
CPUnbr = 4
UserName = COMPUTOWN\John
Password = *****
RemoteDrive = C
RemoteDirectory = John\dynare_calcs\Remote
DynarePath = c:\dynare\matlab
MatlabOctavePath = matlab
```

3.2.5 Example Unix syntax for remote runs

- no `Password` and `RemoteDrive` fields are needed;
- `ComputerName` is the full IP address or the DNS address.

One remote slave: the following command defines remote runs on the machine `name.domain.org`.

```
[cluster]
Name = unix1
Members = n2

[node]
Name = n2
ComputerName = name.domain.org
CPUnbr = 4
UserName = JohnSmith
RemoteDirectory = /home/john/Remote
DynarePath = /home/john/dynare/matlab
MatlabOctavePath = matlab
```

Combining local and remote runs: the following commands define a cluster of local an remote CPU's.

```
[cluster]
Name = unix2
Members = n1 n2

[node]
Name = n1
ComputerName = localhost
CPUnbr = 4

[node]
Name = n2
ComputerName = name.domain.org
CPUnbr = 4
UserName = JohnSmith
RemoteDirectory = /home/john/Remote
DynarePath = /home/john/dynare/matlab
MatlabOctavePath = matlab
```

3.2.6 Testing the cluster

In this section we describe what happens when the user omits a mandatory entry or provides bad values for them and how DYNARE reacts in these cases. In the parallel package there is a utility (`AnalyseComputationalEnvironment.m`) devoted to this task (this is triggered by the command line option `parallel_test`). When necessary during the discussion, we use the `parallel` entries used in the previous examples.

ComputerName: If `Local=0`, DYNARE checks if the computer `vonNeumann` exists and

if it is possible communicate with it. If this is not the case, an error message is generated and the computation is stopped.

CPUnbr: a value for this variable must be in the form `[s:d]` with `d>=s`. If the user types values `s>d`, their order is flipped and a warning message is sent. When the user provides a correct value for this field, DYNARE checks if `d` CPUs (or cores) are available on the computer. Suppose that this check returns an integer `nC`. We can have three possibilities:

1. `nC= d;` all the CPU's available are used, no warning message are generated by DYNARE;
2. `nC> d;` some CPU's will not be used;
3. `nC< d;` DYNARE alerts the user that there are less CPU's than those declared. The parallel tasks would run in any case, but some CPU's will have multiple instances assigned, with no gain in computational time.

UserName & Password: if `Local = 1`, no information about user name and password is necessary: "I am working on this computer". When remote computations on a Windows network are required, DYNARE checks if the user name and password are correct, otherwise execution is stopped with an error; for a Unix network, the user and the proper operation of SSH is checked;

RemoteDrive & RemoteDirectory: if `Local = 1`, these fields are not required since the working directory of the 'slaves' will be the same of the 'master'. If `Local = 0`, DYNARE tries to copy a file (`Tracing.txt`) in this remote location. If this operation fails, the DYNARE execution is stopped with an error;

MatlabOctavePath & DynarePath: MATLAB instances are tried on slaves and the DYNARE path is checked.

3.3 The Developers perspective

In this section we describe with some accuracy the DYNARE parallel routines.

Windows: With Windows operating system, the parallel package requires the installation of a free software package called PsTools (Russinovich, 2009). PsTools suite is a resource kit with a number of command line tools that mimics administrative features available under the Unix environment. PsTools can be downloaded from Russinovich (2009) and extracted in a Windows directory on your computer: to make PsTools working properly, it is mandatory to add this directory to the Windows path. After this step it is possible to invoke and use the PsTools commands from any location in the Windows file system. PsTools, MATLAB and DYNARE have to be installed and work properly on all the machines in the grid for parallel computation.

Unix: With Unix operating system, SSH must be installed on the master and on the slave machines. Moreover, SSH keys must be installed so that the SSH connections from the master to the slaves can be done without passwords.

As soon as the computational environment is set-up for working on a grid of CPU's, the parallel package allows to parallelize any loop that is computationally expensive, following the step by step procedure showed in Table 1. This is done using five basic functions: `masterParallel.m`, `fParallel.m` or `slaveParallel.m`, `fMessageStatus.m`, `closeSlave.m`.

`masterParallel` is the entry point to the parallelization system:

- It is called from the master computer, at the point where the parallelization system should be activated. Its main arguments are the name of the function containing the task to be run on every slave computer, inputs to that function stored in two structures (one for local and the other for global variables), and the configuration of the cluster; this function exits when the task has finished on all computers of the cluster, and returns the output in a structure vector (one entry per slave);
- all file exchange through the filesystem is concentrated in this `masterParallel` routine: so it prepares and send the input information for slaves, it retrieves

from slaves the info about the status of remote computations stored on remote slaves by the remote processes; finally it retrieves outputs stored on remote machines by slave processes;

- there are two modes of parallel execution, triggered by option `parallel_slave_open_mode`:
  - when `parallel_slave_open_mode=0`, the slave processes are closed after the completion of each task, and new instances are initiated when a new job is required; this mode is managed by `fParallel.m` ['Open-Close'];
  - when `parallel_slave_open_mode=1`, the slave processes are kept running after the completion of each task, and wait for new jobs to be performed; this mode is managed by `slaveParallel.m` ['Always-Open'];

`slaveParallel.m/fParallel.m`: are the top-level functions to be run on every slave; their main arguments are the name of the function to be run (containing the computing task), and some information identifying the slave; the functions use the input information that has been previously prepared and sent by `masterParallel` through the filesystem, call the computing task, finally the routines store locally on remote machines the outputs such that `masterParallel` retrieves back the outputs to the master computer;

`fMessageStatus.m`: provides the core for simple message passing during slave execution: using this routine, slave processes can store locally on remote machine basic info on the progress of computations; such information is retrieved by the master process (i.e. `masterParallel.m`) allowing to echo progress of remote computations on the master; the routine `fMessageStatus.m` is also the entry-point where a signal of interruption sent by the master can be checked and executed; this routine typically replaces calls to `waitbar.m`;

`closeSlave.m` is the utility that sends a signal to remote slaves to close themselves. In the standard operation, this is only needed with the 'Always-Open' mode and it is called when DYNARE computations are completed. At that point, `slaveParallel.m` will get a signal to terminate and no longer wait for new jobs. However, this utility is

also useful in any parallel mode if, for any reason, the master needs to interrupt the remote computations which are running;

The parallel toolbox also includes a number of utilities:

- `AnalyseComputationalEnviroment.m`: this a testing utility that checks that the cluster works properly and echoes error messages when problems are detected;
- `InitializeComputationalEnviroment.m` : initializes some internal variables and remote directories;
- `distributeJobs.m`: uses a simple algorithm to distribute evenly jobs across the available CPU's;
- a number of generalized routines that properly perform `delete`, `copy`, `mkdir`, `rmdir` commands through the network file-system (i.e. used from the master to operate on slave machines); the routines are adaptive to the actual environment (Windows or Unix);

  `dynareParallelDelete.m` : generalized `delete`;

  `dynareParallelDir.m` : generalized `dir`;

  `dynareParallelGetFiles.m` : generalized `copy` FROM slaves TO master machine;

  `dynareParallelMkDir.m` : generalized `mkdir` on remote machines;

  `dynareParallelRmDir.m` : generalized `rmdir` on remote machined;

  `dynareParallelSendFiles.m` : generalized `copy` TO slaves FROM master machine;

In Table 1 we have synthesized the main steps for parallelizing MATLAB codes.

So far, we have parallelized the following functions, by selecting the most computationally intensive loops:

1. the cycle looping for multiple chain random walk Metropolis:

   `random_walk_metropolis_hastings`,

   `random_walk_metropolis_hastings_core`;

1. locate within DYNARE the portion of code suitable to be parallelized, i.e. an expensive cycle `for`;

2. suppose that the function `tuna.m` contains a cycle `for` that is suitable for parallelization: this cycle has to be extracted from `tuna.m` and put it in a new MATLAB function named `tuna_core.m`;

3. at the point where the expensive cycle should start, the function `tuna.m` invokes the utility `masterParallel.m`, passing to it the `options_.parallel` structure, the name of the of the function to be run in parallel (`tuna_core.m`), the local and global variables needed and all the information about the files (MATLAB functions `*.m`; data files `*.mat`) that will be handled by `tuna_core.m`;

4. the function `masterParallel.m` reads the input arguments provided by `tuna.m` and:

   - decides how to distribute the task evenly across the available CPU's (using the utility routine `distributeJobs.m`); prepares and initializes the computational environment (i.e. copy files/data) for each slave machine;
   - uses the PsTools and the Operating System commands to launch new MATLAB instances, synchronize the computations, monitor the progress of slave tasks through a simple message passing system (see later) and collect results upon completion of the slave threads;

5. the slave threads are executed using the MATLAB functions `fParallel.m`/`slaveParallel.m` as wrappers for implementing the tasks sent by the master (i.e. to run the `tuna_core.m` routine);

6. the utility `fMessageStatus.m` can be used within the core routine `tuna_core.m` to send information to the master regarding the progress of the slave thread;

7. when all DYNARE computations are completed, `closeSlave.m` closes all open remote MATLAB/OCTAVE instances waiting for new jobs to be run.

Table 1: Procedure for parallelizing portions of codes.

2. the cycle looping for multiple chain independent Metropolis:
   `independent_metropolis_hastings.m`,
   `independent_metropolis_hastings_core.m`;

3. the cycle looping over estimated parameters computing univariate diagnostics:
   `mcmc_diagnostics.m`,
   `mcmc_diagnostics_core.m`;

4. the Monte Carlo cycle looping over posterior parameter subdraws performing the IRF simulations (`<*>_core1`) and the cycle looping over exogenous shocks plotting IRF's charts (`<*>_core2`):
   `posteriorIRF.m`,
   `posteriorIRF_core1.m`, `posteriorIRF_core2.m`;

5. the Monte Carlo cycle looping over posterior parameter subdraws, that computes filtered, smoothed, forecasted variables and shocks:
   `prior_posterior_statistics.m`,
   `prior_posterior_statistics_core.m`;

6. the cycle looping over endogenous variables making posterior plots of filter, smoother, forecasts: `pm3.m`, `pm3_core.m`.

3.3.1 Write a parallel code: an example

Using a MATLAB pseudo (but very realistic) code, we now describe in detail how to use the above step by step procedure to parallelize the random walk Metropolis Hastings algorithm. Any other function can be parallelized in the same way.

It is obvious that most of the computational time spent by the `random_walk_metropolis_hastings.m` function is given by the cycle looping over the parallel chains performing the Metropolis:

```
function random_walk_metropolis_hastings
       (TargetFun, ProposalFun, ..., varargin)
[...]
for b = fblck:nblck,
...
end
[...]
```

Since those chains are totally independent, the obvious way to reduce the computational time is to parallelize this loop, executing the `(nblck-fblck)` chains on different computers/CPUs/cores.

To do so, we remove the `for` cycle and put it in a new function named `<*>_core.m`:

```
function myoutput =
    random_walk_metropolis_hastings_core(myinputs,fblck,nblck, ...)
[...]
```

just list global variables needed (they are set-up properly by `fParallel` or `slaveParallel`)

```
global estim_params_ options_  M_ oo_
```

here we collect all local variables stored in `myinputs`

```
TargetFun=myinputs.TargetFun;
ProposalFun=myinputs.ProposalFun;
xparam1=myinputs.xparam1;
[...]
```

here we run the loop

```
for b = fblck:nblck,
...
end
[...]
```

here we wrap all output arguments needed by the 'master' routine

```
myoutput.record = record;
[...]
```

The split of the `for` cycle has to be performed in such a way that the new `<*>_core` function can work in both serial and parallel mode. In the latter case, such a function will be invoked by the slave threads and executed for the number of iterations assigned by `masterParallel.m`.

The modified `random_walk_metropolis_hastings.m` is therefore:

```
function random_walk_metropolis_hastings(TargetFun,ProposalFun,\ldots,varargin)
[...]
% here we wrap all local variables needed by the <*>_core function
localVars = struct('TargetFun', TargetFun, ...
[...]
    'd', d);
[...]
% here we put the switch between serial and parallel computation:
if isnumeric(options_.parallel) || (nblck-fblck)==0,
% serial computation
    fout = random_walk_metropolis_hastings_core(localVars, fblck,nblck, 0);
    record = fout.record;

else
% parallel computation

    % global variables for parallel routines
    globalVars = struct('M_',M_, ...
                        [...]
                        'oo_', oo_);

    % which files have to be copied to run remotely
    NamFileInput(1,:) = {'',[ModelName '_static.m']};
    NamFileInput(2,:) = {'',[ModelName '_dynamic.m']};
    [ ...]

    % call the master parallelizing utility
    [fout, nBlockPerCPU, totCPU] = masterParallel(options_.parallel, ...
    fblck, nblck, NamFileInput, 'random_walk_metropolis_hastings_core',
    localVars, globalVars, options_.parallel_info);

    % collect output info from parallel tasks provided in fout
    [ ...]
end

% collect output info from either serial or parallel tasks
irun = fout(1).irun;
NewFile = fout(1).NewFile;
[...]
```

Finally, in order to allow the master thread to monitor the progress of the slave threads, some message passing elements have to be introduced in the `<*>_core.m` file. The utility function `fMessageStatus.m` has been designed as an interface for this task, and can be seen as a generalized form of the MATLAB utility `waitbar.m`.

In the following example, we show a typical use of this utility, again from the random walk Metropolis routine:

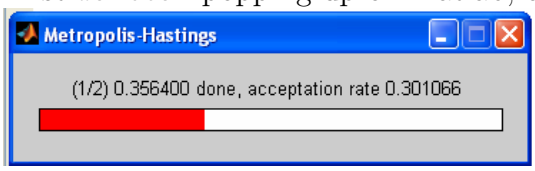
```
for  j = 1:nruns
[...]
% define the progress of the loop:
prtfrc = j/nruns;

% define a running message:
% first indicate which chain is running on the current CPU [b]
% out of the chains [mh_nblock] requested by the DYNARE user
waitbarString = [ '(' int2str(b) '/' int2str(mh_nblck) ') ...

% then add possible further information, like the acceptation rate
    ' sprintf('%f done, acceptation rate %f',prtfrc,isux/j)]

    if mod(j, 3)==0 & ~whoiam
        % serial computation
        waitbar(prtfrc,hh_fig,waitbarString);

    elseif mod(j,50)==0 & whoiam,
        % parallel computation
        fMessageStatus(prtfrc, ...
            whoiam, ...
            waitbarString, ...
            waitbarTitle, ...
            options_.parallel(ThisMatlab))

    end
    [...]
end
```

In the previous example, a number of arguments are used to identify which CPU and which computer in the claster is sending the message, namely:

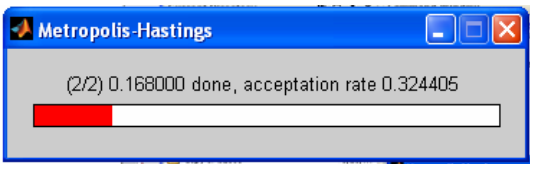
```
%  whoiam [int]         index number of this CPU among all CPUs in the
%                       cluster
%  ThisMatlab [int]     index number of this slave machine in the cluster
%                       (entry in options_.parallel)
```

The message is stored as a MATLAB data file `*.mat` saved on the working directory of remote slave computer. The master will will check periodically for those messages and retrieve the files from remote computers and produce an advanced monitoring plot.

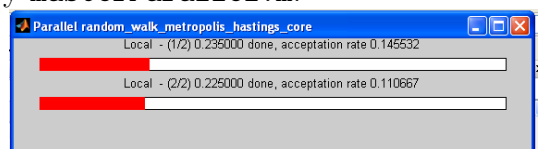
So, assuming to run two Metropolis chains, under the standard serial implementation there will be a first `waitbar` popping up on matlab, corresponding to the first chain:



followed by a second `waitbar`, when the first chain is completed.



On the other hand, under the parallel implementation, a parallel monitoring plot will be produced by `masterParallel.m`:



4 Parallel DYNARE: testing

We checked the new parallel platform for DYNARE performing a number of tests, using different models and computer architectures. We present here all tests performed with Windows XP/MATLAB. However, similar tests were performed successfully under Linux/Ubuntu environment. In the Bayesian estimation of DSGE models with DYNARE, most of the computing time is devoted to the posterior parameter estimation with the Metropolis algorithm. The first and second tests are therefore focused on the parallelization of the Random Walking Metropolis Hastings algorithm (Sections 4.1-4.2). In addition, further tests (Sections 4.3-4.4) are devoted to test all the parallelized functions in DYNARE.

4.1 Test 1.

The main goal here was to evaluate the parallel package on a *fixed hardware platform* and using chains of *variable length*. The model used for testing is a modification of Hradisky et al. (2006). This is a small scale open economy DSGE model with 6 observed variables, 6 endogenous variables and 19 parameters to be estimated. We estimated the model on a bi-processor machine (Fujitsu Siemens, Celsius R630) powered with an Intel® Xeon™CPU 2.80GHz Hyper Treading Technology; first with the original serial Metropolis and subsequently using the parallel solution, to take advantage of the two processors technology. We ran chains of increasing length: 2500, 5000, 10,000, 50,000, 100,000,

250,000, 1,000,000.

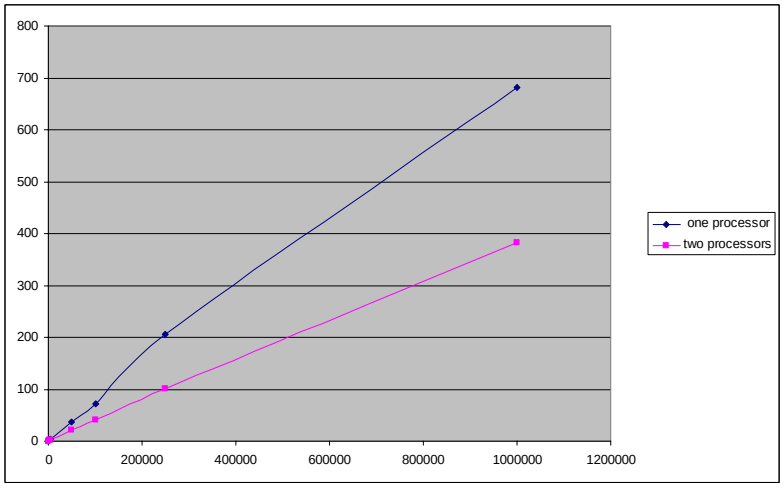


Figure 1: Computational time (in minutes) versus chain length for the serial and parallel implementation (Metropolis with two chains).

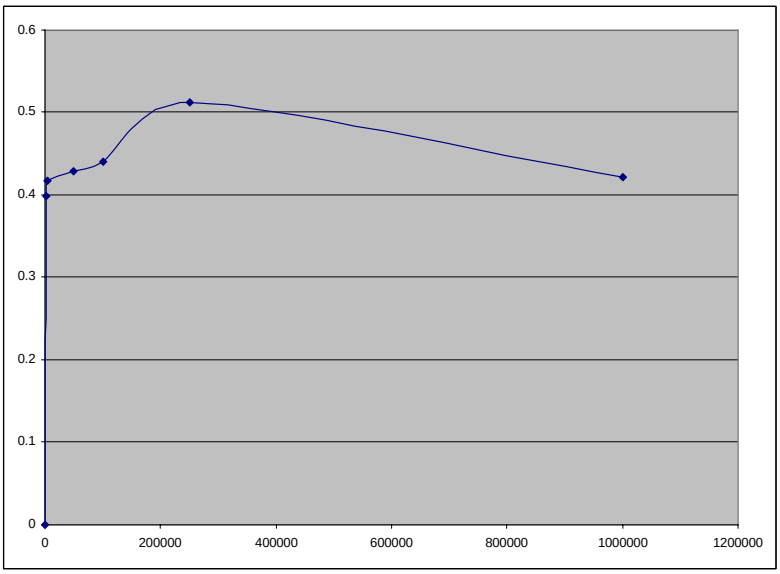


Figure 2: Reduction of computational time (i.e. the 'time gain') using the parallel coding versus chain length. The time gain is computed as $(T_s - T_p)/T_p$, where T_s and T_p denote the computing time of the serial and parallel implementations respectively.

Overall results are given in Figure 1, showing the computational time versus chain length, and Figure 2, showing the reduction of computational time (or the time gain) with respect to the serial implementation provided by the parallel coding. The gain in computing time of the exercise is of about 45% on this test case, so reducing from 11.40 hours to about 6 hours the cost of running 1,000,000 Metropolis iterations (the ideal gain would be of 50% in this case).

Machine	Single-processor	Bi-processor	Dual core
Parallel	8:01:21	7:02:19	5:39:38
Serial	10:12:22	13:38:30	11:02:14
Speed-Up rate	1.2722	1.9381	1.9498
Ideal Speed-UP rate	∼1.5	2	2

Table 2: Trail results with normal PC operation. Computing time expressed in h:m:s. Speed-up rate is computed as T_s/T_p, where T_s and T_p are the computing times for the serial and parallel implementations.

4.2 Test 2.

The scope of the second test was to verify if results were robust over different hardware platforms. We estimated the model with chain lengths of 1,000,000 runs on the following hardware platforms:

- Single processor machine: Intel® Pentium4® CPU 3.40GHz with Hyper Treading Technology (Fujitsu-Siemens Scenic Esprimo);
- Bi-processor machine: two CPU's Intel® Xeon™2.80GHz Hyper Treading Technology (Fujitsu-Siemens, Celsius R630);
- Dual core machine: Intel Centrino T2500 2.00GHz Dual Core (Fujitsu-Siemens, LifeBook S Series).

We first run the tests with normal configuration. However, since (i) dissimilar software environment on the machine can influence the computation; (ii) Windows service (Network, Hard Disk writing, Demon, Software Updating, Antivirus, etc.) can start during the simulation; we also run the tests not allowing for any other process to start during the estimation. Table 2 gives results for the ordinary software environment and process priority is set as low/normal.

Results showed that Dual-core technology provides a similar gain if compared with bi-processor results, again about 45%. The striking results was that the Dual-core processor clocked at 2.0GHz was about 30% faster than the Bi-processor clocked at 2.8GHz. Interesting gains were also obtained via multi-threading on the Single-processor machine, with speed-up being about 1.27 (i.e. time gain of about 21%). However, beware that we burned a number of processors performing tests on single processors with hyper-threading

Environment	Computing time	Speed-up rate w.r.t. Table 2
Parallel Waitbar Not Visible	5:06:00	1.06
Parallel waitbar Not Visible, Real-time Process priority, Unplugged network cable.	4:40:49	1.22

Table 3: Trail results with different software configurations (optimized operating environment for computational requirements).

and using very long chains (1,000,000 runs)! We re-run the tests on the Dual-core machine, by cleaning the PC operation from any interference by other programs and show results in Table 3. A speed-up rate of 1.06 (i.e. 5.6% time gain) can be obtained simply hiding the MATLAB waitbar. The speed-up rate can be pushed to 1.22 (i.e. 18% time gain) by disconnecting the network and setting the priority of the process to real time. It can be noted that from the original configuration, taking 11:02 hours to run the two parallel chains, the computational time can be reduced to 4:40 hours (i.e. for a total time gain of over 60% with respect to the serial computation) by parallelizing and optimally configuring the operating environment. These results are somehow surprising and show how it is possible to reduce dramatically the computational time with slight modification in the software configuration.

Given the excellent results reported above, we have parallelized many other DYNARE functions. This implies that parallel instances can be invoked many times during a single DYNARE session. Under the basic parallel toolbox implementation, that we call the 'Open/Close' strategy, this implies that MATLAB instances are opened and closed many times by system calls, possibly slowing down the computation, specially for 'entry-level' computer resources. As mentioned before, this suggested to implement an alternative strategy for the parallel toolbox, that we call the 'Always-Open' strategy, where the slave MATLAB threads, once opened, stay alive and wait for new tasks assigned by the master until the full DYNARE procedure is completed. We show next the tests of these latest implementations.

4.3 Test 3

In this Section we use the Lubik (2003) model as test function[2] and a very simple computer class, quite diffuse nowadays: Netbook personal Computer. In particular we used the Dell Mini 10 with Processor Intel® Atom™Z520 (1,33 GHz, 533 MHz), 1 GB di RAM (with Hyper-trading). First, we tested the computational gain of running a full Bayesian estimation: Metropolis (two parallel chains), MCMC diagnostics, posterior IRF's and filtered, smoothed, forecasts, etc. In other words, we designed DYNARE sessions that invoke all parallelized functions. Results are shown in Figures 3-4. In Figure 3 we show the computational time versus the length of the Metropolis chains in the serial and parallel setting ('Open/Close' strategy). With very short chain length, parallel setting obviously slows down performances of the computations (due to delays in open/close MATLAB sessions and in synchronization), while increasing the chain length, we can get speed-up rates up to 1.41 on this 'entry-level' portable computer (single processor and Hyper-threading). In order to appreciate the gain of parallelizing all functions invoked after Metropolis, in Figure 4 we show the results of the experiment, but without running Metropolis, i.e. we use the options `load_mh_files = 1` and `mh_replic = 0` DYNARE options (i.e. Metropolis and MCMC diagnostics are not invoked). The parallelization of the functions invoked after Metropolis allows to attain speed-up rates of 1.14 (i.e. time gain of about 12%). Note that the computational cost of these functions is proportional to the chain length only when the latter is relatively small. In fact, the number of sub-draws taken by `posteriorIRF.m` or `prior_posterior_statistics.m` is proportional to the total number of MH draws up to a maximum threshold of 500 sub-draws (for IRF's) and 1,200 sub-draws (for smoother). This is reflected in the shape of the plots, which attain a plateau when these thresholds are reached. In Figures 5-6 we plot results of the same type of tests just described, but comparing the 'Open/Close' and the 'Always-open' strategies. We can see in both graphs that the more sophisticated approach 'Always-open' provides some reduction in computational time. When the entire Bayesian analysis is performed (including Metropolis and MCMC diagnostics, Figure 5) the gain is on average of 5%,

[2]The Lubik (2003) model is also selected as the 'official' test model for the parallel toolbox in DYNARE.

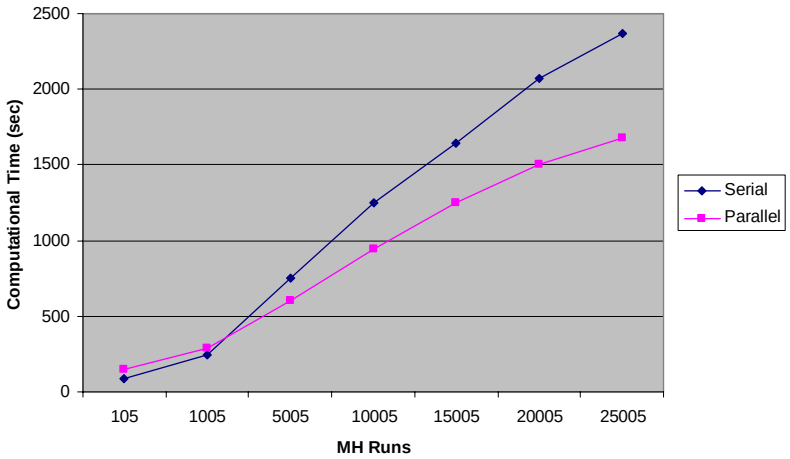


Figure 3: Computational Time (s) versus Metropolis length, running all the parallelized functions in DYNARE and the basic parallel implementation (the 'Open/Close' strategy). (Lubik, 2003).

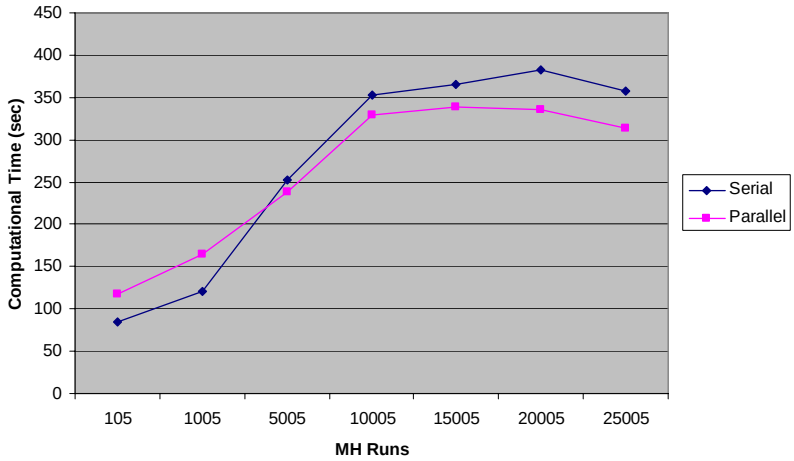


Figure 4: Computational Time (s) versus Metropolis length, loading previously performed MH runs and running *only* the parallelized functions after Metropolis (Lubik, 2003). Basic parallel implementation (the 'Open/Close' strategy).

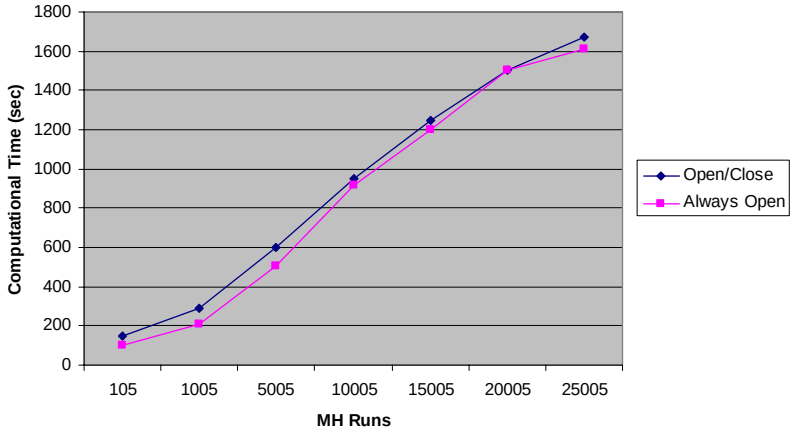


Figure 5: Comparison of the 'Open/Close' strategy and the 'Always-open' strategy. Computational Time (s) versus Metropolis length, running all the parallelized functions in DYNARE (Lubik, 2003).

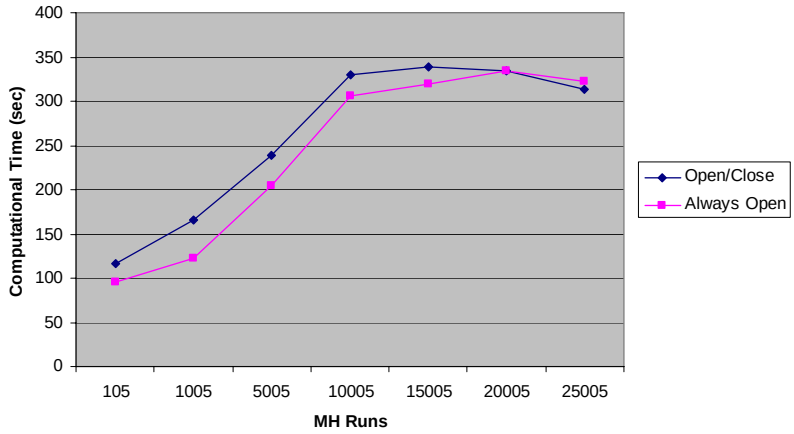


Figure 6: Comparison of the 'Open/Close' strategy and the 'Always-open' strategy. Computational Time (s) versus Metropolis length, running only the parallelized functions after Metropolis (Lubik, 2003).

but it can be more than 10% for short chains. When the Metropolis is not performed, the gain rises on average at about 10%. As expectable, the gain of the 'Always-open' strategy is specially visible when the computational time spent in a single parallel session is not too long if compared to the cost of opening and closing new MATLAB sessions under the 'Open/Close' approach.

4.4 Test 4

Here we increase the dimension of the test model, using the QUEST III model (Ratto et al., 2009), using a more powerful Notebook Samsung Q 45 with an Dual core Processor Intel Centrino. In Figures 7-8 we show the computational gain of the parallel coding with the 'Open/Close' strategy. When the Metropolis is included in the analysis (Figure 7), the computational gain increases with the chain length. For 50,000 MH iterations, the speed-up rate is about 1.42 (i.e. a 30% time gain), but pushing the computation up to 1,000,000 runs provides an almost ideal speed-up of 1.9 (i.e. a gain of about 50% similar to Figure 1). It is also interesting to note that for this medium/large size model, even at very short chain length, the parallel coding is always winning over the serial. Excluding the Metropolis from DYNARE execution (Figure 8), we can see that the speed-up rate of running the posterior analysis in parallel on two cores reaches 1.6 (i.e. 38% of time gain).

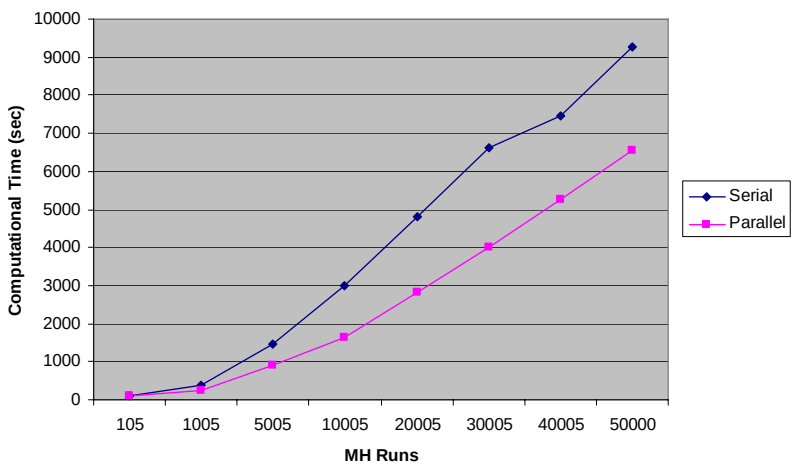


Figure 7: Computational Time (s) versus Metropolis length, running all the parallelized functions in DYNARE and the basic parallel implementation (the 'Open/Close' strategy). (Ratto et al., 2009).

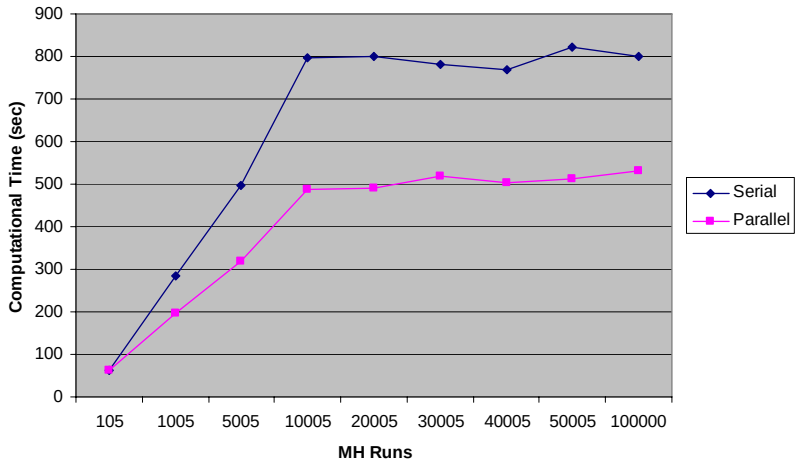


Figure 8: Computational Time (s) versus Metropolis length, loading previously performed MH runs and running *only* the parallelized functions after Metropolis (Ratto et al., 2009). Basic parallel implementation (the 'Open/Close' strategy).

We also checked the efficacy of the 'Always-open' approach with respect to the 'Open/Close' (Figures 9 and 10). We can see in Figure 9 that, running the entire Bayesian analysis, no advantage can be appreciated from the more sophisticated 'Always-open' approach.

On the other hand, in Figure 10, we can see that the 'Always-open' approach still provides a small speed-up rate of about 1.03. These results confirm the previous comment that the gain of the 'Always-open' strategy is specially visible when the computational time spent in a single parallel session is not too long, and therefore, the bigger the model size, the less the advantage of this strategy.

5 Conclusions

The methodology identified for parallelizing MATLAB codes within DYNARE proved to be effective in reducing the computational time of the most extensive loops. This methodology is suitable for 'embarrassingly parallel' codes, requiring only a minimal communication flow between slave and master threads. The parallel DYNARE is built around a few 'core' routines, that act as a sort of 'parallel paradigm'. Based on those routines, parallelization of expensive loops is made quite simple for DYNARE developers. A basic message passing system is also provided, that allows the master thread to monitor the progress of slave threads. The test model `ls2003.mod` is available in the folder

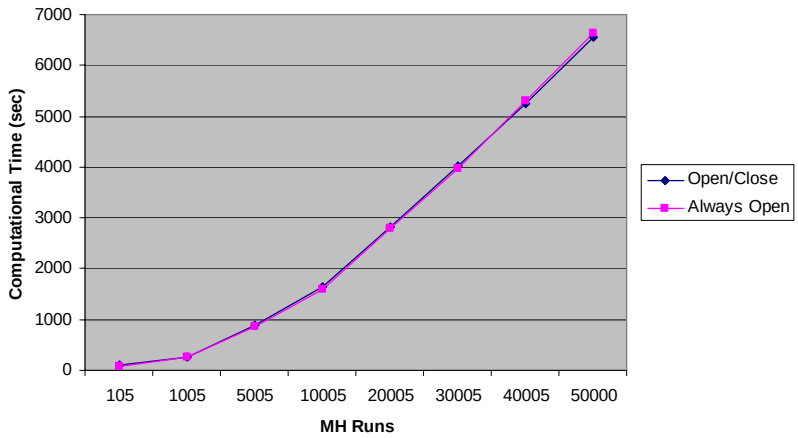


Figure 9: Comparison of the 'Open/Close' strategy and the 'Always-open' strategy. Computational Time (s) versus Metropolis length, running all the parallelized functions in DYNARE (Ratto et al., 2009).

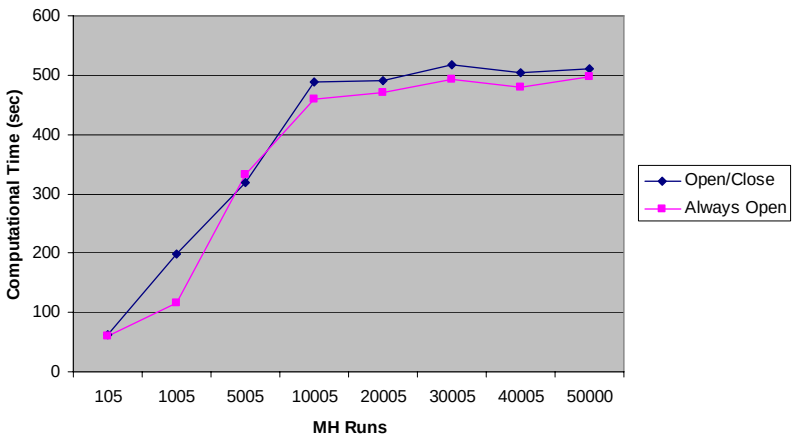


Figure 10: Comparison of the 'Open/Close' strategy and the 'Always-open' strategy. Computational Time (s) versus Metropolis length, running only the parallelized functions after Metropolis (QUEST III model Ratto et al., 2009).

`\tests\parallel` of the DYNARE distribution, that allows running parallel examples.

References

I. Azzini, R. Girardi, and M. Ratto. Parallelization of Matlab codes under Windows platform for Bayesian estimation: A DYNARE application. In *DYNARE CONFERENCE*, Paris School of Economics, September 2007.

B Barney. *Introducing To Parallel Computing (Tutorial)*. Lawrence Livermore National Laboratory, 2009.

J.G. Brookshear. *Computer Science: An Overview*. Pearson, 10th edition, 2009. ISBN 0-321-52403-9. http://www.aw-bc.com/brookshear/.

W.L. Goffe and M. Creel. Multi-core CPUs, clusters, and grid computing: A tutorial. *Computational economics*, 32(4):353–382, 2008.

M. Hradisky, R. Liska, M. Ratto, and R. Girardi. Exchange Rate Versus Inflation Targeting in a Small Open Economy SDGE Model, for European Union New Members States. In *DYNARE CONFERENCE, Paris September 4-5*, 2006.

T. Lubik. Investment spending, equilibrium indeterminacy, and the interactions of monetary and fiscal policy. Technical Report Economics Working Paper Archive 490, The Johns Hopkins University, 2003.

Marco Ratto, Werner Roeger, and Jan in 't Veld. QUEST III: An estimated open-economy DSGE model of the euro area with fiscal and monetary policy. *Economic Modelling*, 26(1):222 – 233, 2009. doi: DOI:10.1016/j.econmod.2008.06.014. URL http://www.sciencedirect.com/science/article/B6VB1-4TC8J5F-1/2/7f22da17478841ac5d7a77d06f13d13e.

M. Russinovich. *PsTools v2.44*, 2009. available at Microsoft TechNet, http://technet.microsoft.com/en-us/sysinternals/bb896649.aspx.

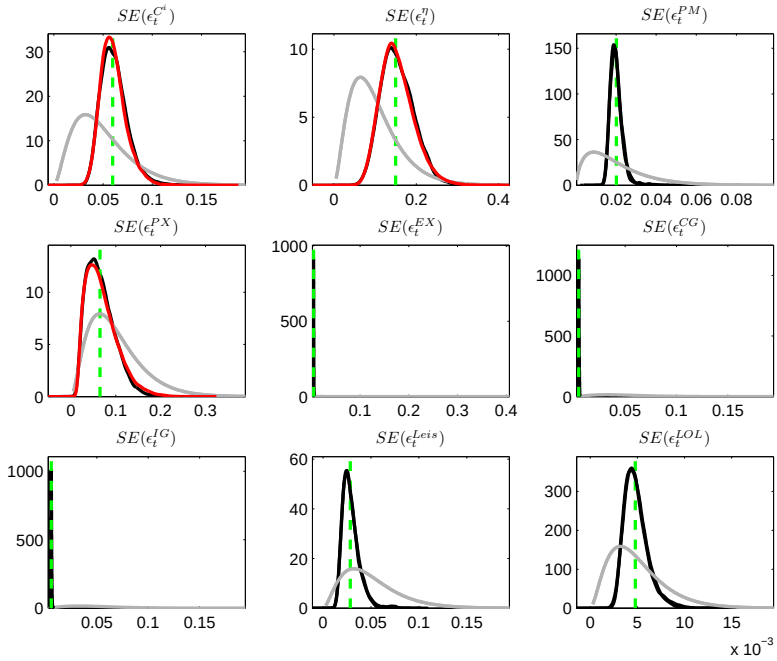


Figure 11: Prior (grey lines) and posterior density of estimated parameters (black = 100,000 runs; red = 1,000,000 runs) using the RWMH algorithm (QUEST III model Ratto et al., 2009).

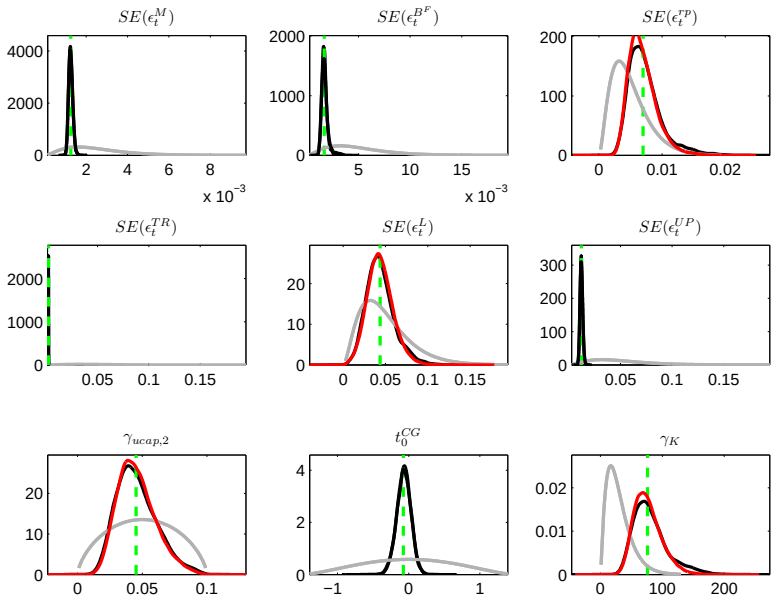


Figure 12: Prior (grey lines) and posterior density of estimated parameters (black = 100,000 runs; red = 1,000,000 runs) using the RWMH algorithm (QUEST III model Ratto et al., 2009).

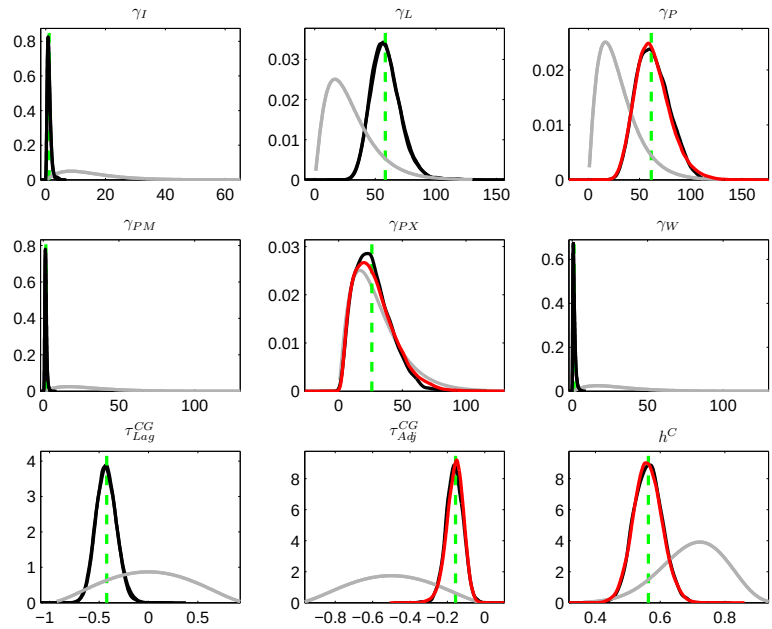


Figure 13: Prior (grey lines) and posterior density of estimated parameters (black = 100,000 runs; red = 1,000,000 runs) using the RWMH algorithm (QUEST III model Ratto et al., 2009).

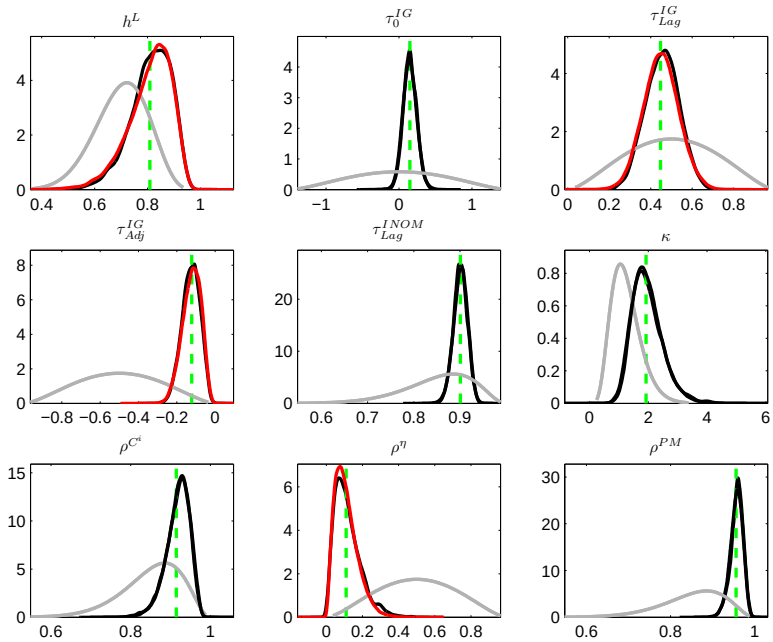


Figure 14: Prior (grey lines) and posterior density of estimated parameters (black = 100,000 runs; red = 1,000,000 runs) using the RWMH algorithm (QUEST III model Ratto et al., 2009).

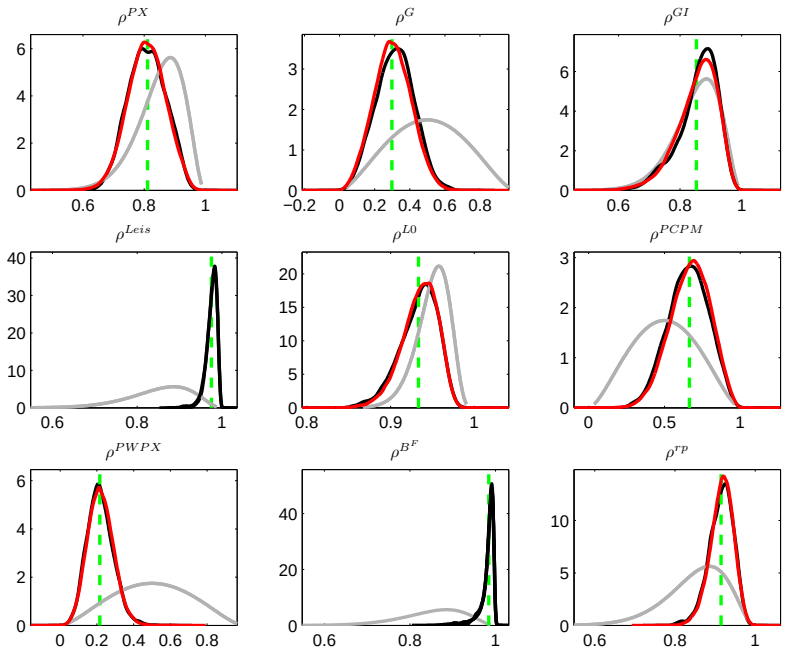


Figure 15: Prior (grey lines) and posterior density of estimated parameters (black = 100,000 runs; red = 1,000,000 runs) using the RWMH algorithm (QUEST III model Ratto et al., 2009).

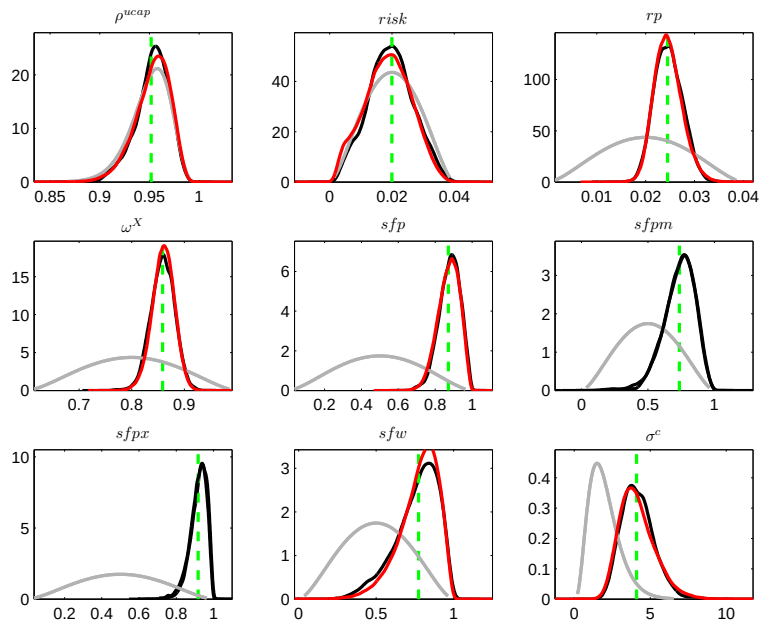


Figure 16: Prior (grey lines) and posterior density of estimated parameters (black = 100,000 runs; red = 1,000,000 runs) using the RWMH algorithm (QUEST III model Ratto et al., 2009).

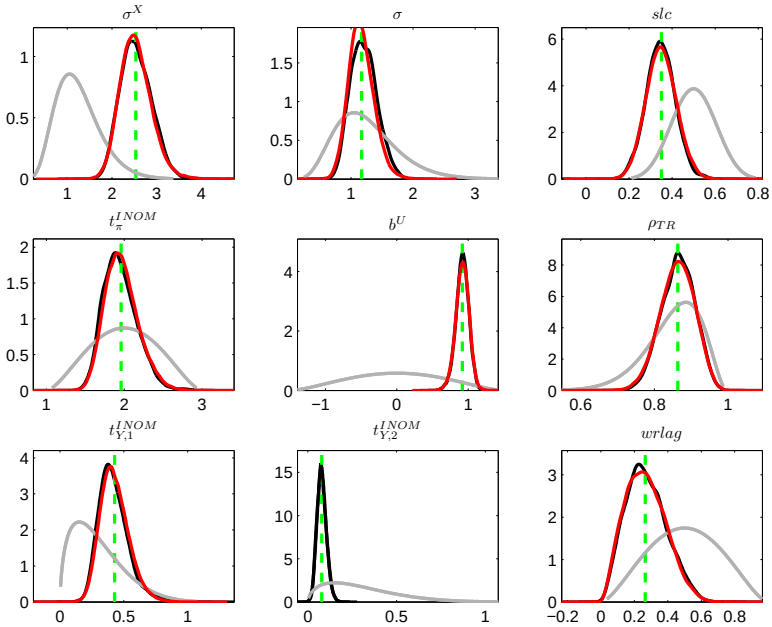


Figure 17: Prior (grey lines) and posterior density of estimated parameters (black = 100,000 runs; red = 1,000,000 runs) using the RWMH algorithm (QUEST III model Ratto et al., 2009).

A A tale on parallel computing

This is a general introduction to Parallel Computing. Readers can skip it, provided they have a basic knowledge of DYNARE and Computer Programming (Goffe and Creel, 2008; Azzini et al., 2007). There exists an ample scientific literature as well as an enormous quantity of information on the Web, about parallel computing. Sometimes, this amount of information may result ambiguous and confusing in the notation adopted and the description of technologies. Then main the goal here is therefore to provide a very simple introduction to this subject, leaving the reader to Brookshear (2009) for a more extensive and clear introduction to computer science.

Modern computer systems (hardware and software) is conceptually identical to the first computer developed by J. Von Neumann. Nevertheless, over time, hardware, software, but most importantly *hardware & software together* have acquired an ever increasing ability to perform incredibly complex and intensive tasks. Given this complexity, we use to explain the modern computer systems as the "avenue paradigm", that we summarize in the next tale.

Nowadays there is a small but lovely town called "CompuTown". In CompuTown there are many roads, which are all very similar to each other, and also many gardens. The most important road in CompuTown is the Von Neumann Avenue. The first building in Von Neumann Avenue has three floors (this is a *computer system*: PC, workstation, etc.; see Figure 18 and Brookshear (2009)). Floors communicate between them only with a single stair. In each floor there are people coming from the same country, with the same language, culture and uses. People living, moving and interacting with each other in the first and second floor are the *programs* or software agents or, more generally speaking, *algorithms* (see chapters 3, 5, 6 and 7 in Brookshear (2009)). Examples of the latter are the softwares MATLAB, Octave, and a particular program called the *operating system* (Windows, Linux, Mac OS, etc.).

People at the *ground floor* are the transistors, the RAM, the CPU, the hard disk, etc. (i.e. the *Computer Architecture*, see chapters 1 and 2 in Brookshear). People at the *second floor* communicate with people at the *first floor* using the only existing scale (the

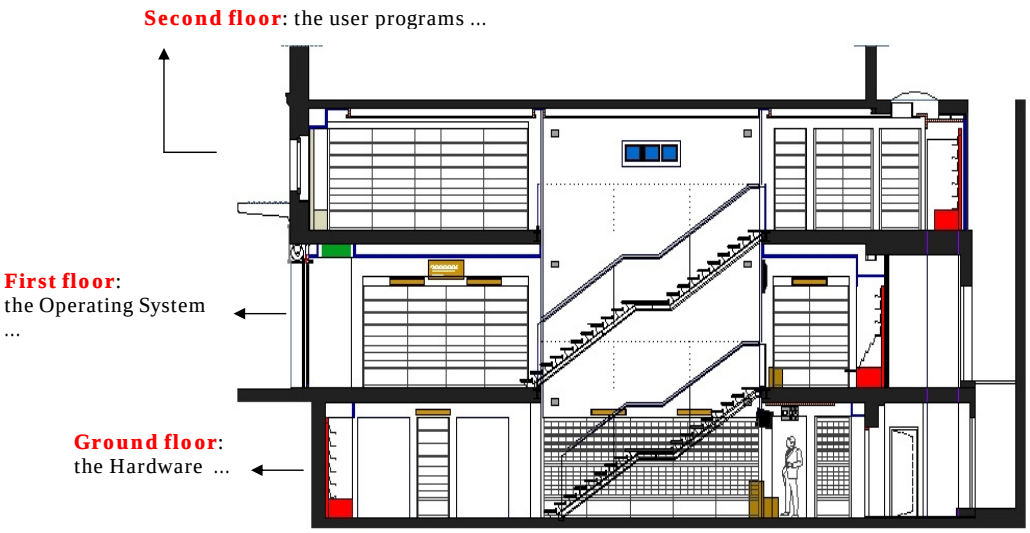


Figure 18: The first building in Von Neumann Avenue: a *Computer System*

pipe). In these communications, people talk two different languages, and therefore do not understand each other. To remove this problem people define a set of words, fixed and understood by everybody: the *Programming Languages*. More specifically, these languages are called *high-level programming languages* (Java, C/C++, FORTRAN,MATLAB, etc.), because they are related to people living on the upper floors of the building! Sometimes people in the building use also pictures to communicate: *the icons* and *graphical user interface*.

In a similar way, people at the first floor communicate with people at the ground floor. Not surprisingly, in this case, people use *low-level programming languages* to communicate to each other (assembler, binary code, machine language, etc.). More importantly, however, people at the first floor must also manage and coordinate the requests from people on the second floor to people at the ground floor, since there is no direct communication between ground and second floor. For example they need to translate high-level programming languages into binary code[3]: the *Operating System* performs this task.

Sometimes, people at the second floor try to talk directly with people at the ground floor, via the *system calls*. In the parallelizing software presented in this document, we will use frequently these system calls, to distribute the jobs between the available hardware

[3]The process to transform an high-level programming languages into binary code is called compilation process.

resources, and to coordinate the overall parallel computational process. If only a single person without family lives on the ground floor, such as the porter, we have a CPU single core. In this case, the porter can only do one task at a time for the people in first or second floor (the main characteristic of the Von Neumann architecture). For example, in the morning he first collects and sorts the mail for the people in the building, and only after completing this task he can take care of the garden. If the porter has to do many jobs, he needs to write in a paper the list of things to do: the *memory* and the *CPU load*. Furthermore, to properly perform its tasks, sometimes the porter has to move some objects trough the passageways at the ground floor (the *System Bus*). If the passageways have standard width, we will have a 32 bits CPU architecture (or bus). If the passageways are very large we will have, for example, a 64 bits CPU architecture (or bus). In this scenario, there will be very busy days where many tasks have to be done and many things have to be moved around: the porter will be very tired, although he will be able to 'survive'. The most afflicted are always the people at the first floor. Every day they have a lot of new, complex requests from the people at the second floor. These requests must be translated in a correct way and passed to the porter. The people at the second floor (the highest floor) "live in cloud cuckoo land". These people want everything to be done easily and promptly: the artificial intelligence, robotics, etc. The activity in the building increases over time, so the porter decides to get helped in order to reduce the execution time for a single job. There are two ways to do this:

- the municipality of CompuTown interconnects all the buildings in the city using roads, so that the porter can share and distribute the jobs (the *Computer Networks*): if the porters involved have the same nationality and language we have a *Computer Cluster*, otherwise we have a *Grid*. Nevertheless, in both cases, it is necessary to define a correct way in which porters can manage, share and complete a shared job: the *communication protocol* (TCP/IP, internet protocol, etc.);

- the building administrator employs an additional porter, producing a *Bi-Processor Computer*. In other case, the porter may get married, producing a *dual-core CPU*. In this case, the wife can help the porter to perform his tasks or even take entirely

some jobs for her (for example do the accounting, take care of the apartment, etc.). If the couple has a children, they can have a further little help: the *thread* and then the *Hyper-threading* technology.

Now a problem arises: who should coordinate the activities between the porters (and their family) and between the other buildings? Or, in other words, should we refurbish the first and second floors to take advantage of the innovations on the ground floor and of the new roads in CompuTown? First we can lodge new persons at the first floor: the operating systems with a set of network tools and multi-processors support, as well as new people at the second floor with new programming paradigms (MPI, OpenMP, Parrallel DYNARE, etc.). Second, a more complex communication scheme between first and ground floor is necessary, building a new set of stairs. So, for example, if we have two stairs between ground and first floor and two porters, using multi-processors and a new parallel programming paradigm, we can assign jobs to each porter directly and independently, and then coordinate the overall work. In parallel DYNARE we use this kind of 'refurbishing' to reduce the computational time and to meet the request of people at the second floor.

Unfortunately, this is only an idealized scenario, where all the citizens in CompuTown live in peace and cooperate between them. In reality, some building occupants argue with each other and this can cause stopping their job: these kinds of conflicts may be linked to *software and hardware compatibility* (between ground and first floor), or to different *software versions* (between second and first floor). The building administration or the municipality of CompuTown have to take care of these problems an fix them, to make the computer system operate properly.

This tale (that can be also called *The Programs's Society*) covered in a few pages the fundamental ideas of computer science.