SEAMLESS MULTICORE PARALLELISM IN MATLAB

Claude Tadonki[1] and Pierre-Louis Caruana[2]
[1]Mines ParisTech - CRI - Mathématiques et Systèmes
35, rue Saint-Honoré, 77305, Fontainebleau Cedex (France)
claude.tadonki@mines-paristech.fr
[2]University of Paris-Sud Orsay
Faculty of Sciences - Bât. 301, 91405 Orsay Cedex (France)
pierre-louis.caruana@u-psud.fr

Abstract

MATLAB is a popular mathematical framework composed of a built-in library implementing a significant set of commonly needed routines. It also provides a language which allows the user to script macro calculations or to write complete programs, hence called "the language of technical computing". So far, a noticeable effort is maintained in order to keep MATLAB being able to cooperate with other standard programming languages or tools. However, this interoperability, which is essential in many circumstances including performance and portability, is not always easy to implement for ordinary scientists. The case of parallel computing is illustrative and needs to be addressed as multicore machines are now standard. In this work, we report our efforts to provide a framework that allow to intuitively express and launch parallel executions within a classical MATLAB code. We study two alternatives, one which is a pure MATLAB solution based on the MATLAB parallel computing toolbox, and another one which implies a symmetric cooperation between MATLAB and C, based on the Pthread library. The later solution does not requires the MATLAB parallel toolbox, thus clearly brings a portability benefit and makes the move to parallel computing within MATLAB less costly to standard users. Experimental results are provided and commented in order to illustrate the use and the efficiency of our solution.

KEY WORDS
MATLAB; parallelism; multicore; threads; speedup; scalability.

1 Introduction

MATLAB is more than a *matrix computation laboratory*, as it covers many kinds of application and provides a well featured programming language. However, as MATLAB users are likely to expect simplicity at all levels of usage, any MATLAB related achievement should fulfill this guideline. The current work related to *multicore parallel programming* is done in that spirit.

Multicore architecture is now the standard for modern processors. This pervasiveness of multiprocessing systems has put a severe pressure on software solutions that can benefit from multicore CPUs [1, 3, 6]. Ordinary users wish to seamlessly harvest the full power of the processor for their basic tasks. Software tools and libraries are now designed accordingly. From the programmer side, its appears that even experts are reluctant to pay so much effort to design multicore parallel codes. Relevant APIs like OpenMP [12] or Cilk [13] were provided to alleviate the programming pain and let the programmer focus on the abstraction model of his code.

MATLAB has earlier started to provide parallel computing solution into its distributions, mainly through additional packages (not provided by default) [5]. From the technical point of view, a certain level of programming skill is still required to implement parallelism within a MATLAB program using native solutions. Therefore, any API designed to hide the underlying effort would be appreciated. This is what we propose in this paper. We first propose a *POSIX-thread* based solution, which thereby drops the need of the MATLAB parallel toolbox. We also explore two alternative APIs that connect the programmer to native MATLAB parallel programming routines. In all cases, the whole process is seamless to the programmer, who just needs to express his parallel execution in a quite intuitive way.

The rest of the paper is organized as follows. The next Section describe native parallelism solutions in MATLAB . Section 3 motivates and provides a detailed description of our contribution. Benchmark results are provided and discussed in Section 5. Section 6 concludes the paper.

2 Overview of existing solutions

First, note that parallelism is provided in recent distributions of MATLAB through additional packages namely *Parallel Computing Toolbox* (PCT) and MATLAB *Distributed Computing Server* (MDCS). In this work, we only focus on multicore parallelism using the *Parallel Computing Toolbox* [10]. This is justified by the fact that (personal) computers are mostly equipped with multicore processor, thus any MATLAB user might think of running tasks in parallel in order to improve performance. We now describe how this is natively provided in recent MATLAB distributions.

2.1 Using parallel built-in libraries

This is the easiest and most seamless method to deal with parallel processing within MATLAB. In fact, in recent (and future) releases of MATLAB , a number of built-in functions are provided through their parallel implementation. Thus, just running a given standard code under a recent version of MATLAB should be sufficient to benefit from the power of parallel processing. While this is really a simple and direct solution, the main drawback is a certain rigidity on the global parallel scheduling. Indeed, the execution model implemented by this approach is the so-called *Locally Parallel Globally Sequential* (LPGS), where parallel execution occurs task by task, in a logical sequential order specified by the programmer. For instance, a parallel version of the *divide-and-conquer* paradigm cannot be carried on with this approach. In addition, not all MATLAB built-in functions are provided with their parallel implementations, the most interesting ones for a given application might be missing. We now describe two ways of dealing with explicit parallelism in MATLAB . In any cases, the parallel language features of MATLAB is enabled through the MATLAB pool. To open the pool, we have to issue the command
`matlabpool open <configuration>`
and to close the pool, which means switch off the parallel features, we issue the command
`matlabpool close`

2.2 MATLAB tasks feature

With this solution, the programmer obtains a parallel execution by creating and submitting several MATLAB tasks to the scheduler. A typical sequence starts with the `createTask` function, which has the following form
`t = createTask(j, F, N, {inputargs}).`
This function creates a new task object into job `j`, and returns a reference to the newly added task object. By this way, several tasks can be added to the same job. Job object `j` is created using the command
`j = createJob(sched)`
where the argument `sched` can be omitted or set to `parcluster()` to use the scheduler identified by the default profile. `F` is the name or handle of the function to be executed within the task, with all its input arguments listed in `{inputargs}` (a row cell array). The function `F` is expected to return `N` outputs that will be retrieved from the job object `j` using the command
`taskoutput = fetchOutputs(j)`
where `taskoutput` is also a row cell array. The programmer is expected to manually extract the outputs from the cell array returns by `fetchOutputs(j)` and put them into the corresponding variables or directly perform the epilogue calculation from it. The example below computes the sum of an array `U` with `8` elements.

```
% Create one job object
  j = createJob();
% Create tasks for job j
  createTask(j, @sum, 1, {U(1:4)});
  createTask(j, @sum, 1, {U(5:8)});
% Submit job j to the scheduler
  submit(j);
% Wait for job completion
  wait(j);
% Get the outputs of job j
  v = fetchOutputs(j);
% Aggregate the partial sums
  s = v{1} + v{2};
% Delete job j
  delete(j);
```

We now state some important facts:

- each task within a job is assigned to a unique MATLAB worker, and is executed independently of the other tasks
- the maximum number of workers is specified in the local scheduler profile, and can be modified as desired, up to a limit of twelve
- if a job has more tasks than allowed workers, the scheduler waits for one of the active tasks to complete before starting another MATLAB worker for the next task. In some cases, such an overloading will prevent the entire job from being executed.

2.3 The parfor construct

The **parfor** statement is used in place of a **for** statement in order to specify that the corresponding loop should be executed in parallel. The loop is therefore split into equal chunks and the corresponding tasks are distributed among the workers. By default, MATLAB uses as many workers as it finds available, unless a more restrictive limit is specified. A typical use of the `parfor` statement is illustrated by the following example with at most 2 workers

```
matlabpool open 2
parfor i=1:10
 U(i) = sqrt(i);
end
matlabpool close
```

The main requirement when using `parfor` is the independence between the iterations. However, if there is only a virtual dependence, i.e. data dependences with no effect on the final result, then MATLAB seems to be able to handle the case correctly. In this particular case, it is important to have a loop which can be executed in any order, like those implementing global reductions. The following script is an example with virtually dependent iterations

```
matlabpool open 2
s = 0;
parfor i=1:10
```

```
 s = s + U(i);
end
matlabpool close
```

The *parfor* feature is more appropriate for "embarrassingly parallel" applications on which some level of performance can be expected. Although quite easy to use, there are number of important facts and restrictions that the programmer should keep in mind when it comes to the *parfor* feature:

- execution of *parfor* is not deterministic in terms of block-iteration order. Thus, we emphasize on having a loop with independent iterations.
- sequential execution might occur in case MATLAB cannot run the `parfor` loop on its pool. This occurs when there is no worker available or if the programmer puts 0 for the parameter specifying the maximum number of workers. Recall that the extended form of the `parfor` construct is as follows
  `parfor(i=1:N,max_workers)`
- *temporary variables* and *loop variables* are treated as local within each chunk of the `parfor` loop. *Sliced variables*, *broadcast variables* and *reduction variables* are treated on a global basis according to the semantic of the loop. Figure 1 illustrates the aforementioned categories of variables.

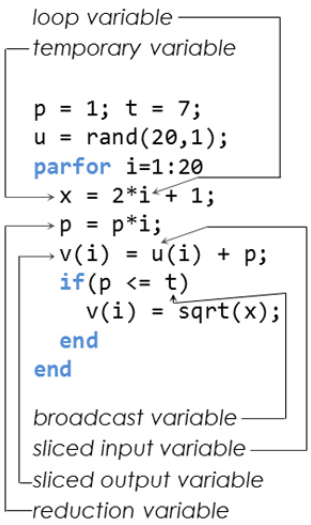

Figure 1. Kinds of variables within a parfor loop

The reader may refer to [9, 2, 4] for more details about MATLAB *parfor* and benchmark oriented discussions.

3 Description of our solutions

3.1 Overview and motivations

Our main goal is to provide a function of the form
`dopar('I1','I2',...,'In');`
where I_i, $i = 1, \cdots, n$ are the instructions to be executed in parallel on a shared memory basis. Each instruction is any valid MATLAB construct that will be executed in the context of the caller, i.e. inputs (resp. outputs) have to be read from (resp. stored to) the caller workspace, which is either a plain script or a function. In future releases, we plan to handle the case I_i is a portion of the MATLAB code, exactly like an OpenMP section. This way of requesting a parallel execution is rather intuitive for any programmer, provided he is aware of the underlying integrity constraints. Only for this reason, and later on for performance needs, it is expected for the user to have some basic prerequisites in multiprocessing, in order to express meaningful parallelism and also to get better scalability. Anyway, the machinery behind is completely seamless to the programmer, which, as usual when it comes to MATLAB , remains focused on the computation rather than programming details.

3.2 Common technical considerations

Since user instructions are provided as *strings*, we use the built-in MATLAB commands *evalin()* [8] to execute the corresponding calculations and *eval()* [7] for assignments between intermediate and input/output variables.

`eval(string)` evaluates the MATLAB code expressed by `string`. Note that this evaluation is performed within the current context. Thus, if this is done within a function, which is our case, we should not expect to directly affect output variables. For input variables, they aren't accessible too, unless passed as arguments to the called function. Let consider the following function to illustrate our case.

```
function my_eval(string)
 eval(string);
end
```

Now, if we issue `my_eval('B = A + 1')`, none of the variables between `A` and `B` will be accessible within the scope of `my_eval`. Instead, they will be treated as local variables with the same names. This is because `A` (resp `B`) is not an input (resp. output) argument of `my_eval`. We will later explain how we address this in the context of the Pthread-based solution.

`evalin(ws, string)` executes the MATLAB code `string` in the context of workspace `ws`, which is either `'base'` for the MATLAB base workspace or `'caller'` for the workspace of the caller function. We use this for pure MATLAB alternatives. The main advantage of `evalin` is that we can directly execute the requested command in the context of the caller program, thus avoiding data import and export.

3.3 Common issues

Whatever the feature we choose to run in the background, the main concern related to variables is that we move into a new context. Consequently, we need to import input data

before executing the requested kernel, and afterward export back output data into the caller context. This is one of the things our framework performs seamlessly, at the price of a certain delay that should be counted in the global time overhead. Moreover, data coming from distinct tasks should be gathered appropriately before updating output variables. This is another postprocessing done by our framework depending on the considered MATLAB parallel feature as we will explain in each of the following sections.

3.4 Pthread based solution

This part is our major contribution as it provides a quite original parallelism solution from the technical point of view. Indeed, with this solution, the programmer does not need any additional MATLAB package, even the *Parallel Computing Toolbox*. The main idea is to use a C code, compiled as a MATLAB *mex-file*, which proceeds as follows:

1. *Parse the associated string* of each input instruction in order to get the list of all involved variables.

2. *Load the data of each right-hand-side variable* from the caller context into the current one.

3. *Launch as many POSIX threads as input instructions*, each thread executes its associated MATLAB instruction using a call to the MATLAB engine [11].

4. *Copy back the data* corresponding to each output variable into the context of the caller.

Figure 2 is an overview of the system, which only requires MATLAB , any C compiler, and the *Pthread* library.

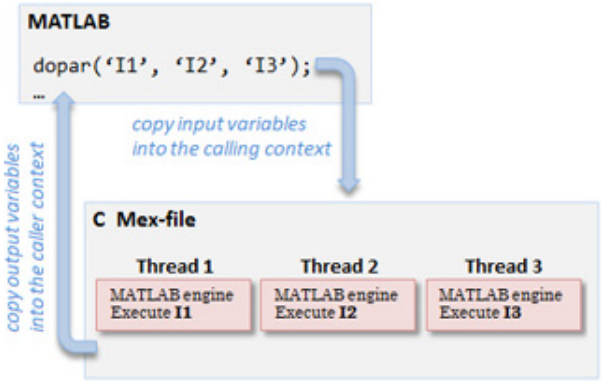

Figure 2. Pthread based architecture

Figure 3 summarizes the commands associated to the MATLAB engine.

Function	Purpose
`engOpen`	Start up MATLAB engine
`engClose`	Shut down MATLAB engine
`engGetVariable`	Get a MATLAB array from the engine
`engPutVariable`	Send a MATLAB array to the engine
`engEvalString`	Execute a MATLAB command
`engOutputBuffer`	Create a buffer to store MATLAB text output
`engOpenSingleUse`	Start a MATLAB engine, nonshared use
`engGetVisible`	Determine visibility of MATLAB engine session
`engSetVisible`	Show or hide MATLAB engine session

Figure 3. Main commands related to MATLAB engines

Let us now comment on each point of the mechanism.

1. **Parsing for variables inventory**. This step is very important, as we need to discriminate between *input variables* and *output variables*. Because the calculations are local, thus out of the context of the caller, we create a local mirror for each variable, and the instruction string is transformed accordingly. For instance, the instruction string `'A = B + C'` is transformed into `'A_c = B_c + C_c'`, where `A_c`, `B_c`, and `C_c` are mirrors of `A`, `B`, and `C` respectively. `B` and `C` are considered as input variables, while `A` is treated as an output variable.

2. **Importing input data**. This is done using the `engGetVariable` routine. The data for each input variable is copied into the associated local mirror.

3. **Threads and MATLAB engine**. Each thread opens a MATLAB engine and issues the execution of its associated instruction string using the `engEvalString` command. Unfortunately, things are not so simple. Indeed, `engOpen` starts a new MATLAB session with a new MATLAB environment. Thus, in addition to the cost of launching a new MATLAB , we need again to explicitly exchange data. One way to avoid this is to use the `/Automation` mode (only available on Windows platforms), which connects to the existing MATLAB session instead of starting a new one. Unfortunately, since we are using a unique MATLAB engine, the threads will have their `engEvalString` commands serialized. This creates a *virtual parallelism*, but not an effective one. We found that the way to go is to use the `engOpenSingleUse`, which starts a non shared MATLAB session, even if we are on the `Automation` mode. The main advantage now, using the `Automation` mode, is that data exchanges can be done by direct assignments (i.e. `engEvalString('A_c = A')` for instance). On a *Linux* platform, we simply use `engOpen` and explicitly exchange data between the different running MATLAB sessions.

4. **Exporting output data**. This is done using the `engPutVariable` routine. Each output variable is assigned the data from its local mirror.

We now explore two alternatives based on native MATLAB implementation of parallel executions.

3.5 MATLAB workers based solution

This solution is based on MATLAB tasks feature as described in Section 2.2. Each instruction string is passed to a worker through the corresponding argument of the `createTask` routine. We start with a MATLAB function `gen_eval`, which can execute any instruction string through the `eval` command. For each instruction string `'I'`, we execute `createTask(job, @gen_eval, 1, 'I');`. Upon completion of all tasks, we retrieve the (aggregated) output data and scatter it according to the set of output variables identified by the parsing. In order to avoid data import and export, due to the fact we are dealing with different contexts, the MATLAB code that creates and runs the tasks is generated on the fly as a string and then executed through the `evalin` routine directly in the context of the caller.

3.6 MATLAB parfor based solution

We link any parallel execution to the *parfor* construct as described in Figure 4. The rest of the mechanism is similar to the MATLAB workers based solution.

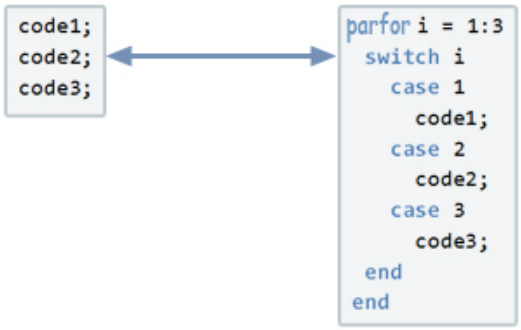

Figure 4. The bridge to the *parfor*

The user should be aware of the restrictions that apply to the use of the *parfor*. One of them is the strong limitation of the number of parallel blocks. Another one is the strict independence between the iterations.

4 Potential performance issues

With the *Pthread* based mechanism, we could reasonably expect a noticeable time overhead because each thread opens a MATLAB session. Data import/export is another potential source of time delay. Other factors that are inherent to shared memory parallelism should be taken into account too (false sharing, serialisation, bus contention, to name a few). All these facts justify the common recommendation of considering parallel execution for performance, only with heavy computing tasks. Table 1 and 2 provide an experimental measurement of the total overhead for each of our three solutions on a 2 cores processor (Intel Core2 Duo Processor E4500). For each solution, the measurements are the costs of the mechanism without any computation nor data transfer. We see form here that the time overhead associated to the *Pthread* solution is the lowest (although the difference with the *task* based solution looks rather marginal).

run	pthread(s)	task(s)	parfor(s)
1	6.8228	5.9950	9.4820
2	4.9977	5.9581	9.4874
3	5.9762	5.9286	9.0390
4	4.9950	5.9685	8.9879
5	4.9103	5.9410	9.0397

Table 1. Pure overhead of our mechanism

l_{vector}	pthread(s)	task(s)	parfor(s)
10^6	0.144	0.687	0.122
2×10^6	0.640	1.407	0.898
3×10^6	0.946	2.114	1.607
4×10^6	1.332	3.777	2.205
5×10^6	1.713	6.604	2.413

Table 2. Time costs for data import&export

We now provide and comment full benchmark results.

5 Illustration and benchmark results

We consider two applications for our benchmark, *sorting* and *matrix-product*. The aim here is to show that our framework is effective in the sense of providing a straightforward way to express and get parallelism under MATLAB . Therefore, the reader should focus on speedup rather than absolute performance. Another point to keep in mind is that the *Pthread*-based solution is our main contribution, thus should be somehow compared with alternatives that are based on pure MATLAB parallel solutions (i.e. *tasks* and *parfor*), although we did the interfacing work for them too.

For *sorting*, we use a MATLAB implementation of the *quicksort* algorithm. On a p cores machine, we use our framework to issue p *quicksort* in parallel, each of them operating on the corresponding chunk of the global array. We also test $2p$ parallel executions when *hyperthreading* is available. For each test, the size provided is the size of the parallel subtask, this should be multiplied by the number of parallel executions to get the global size of the main problem.

For *matrix-product*, we apply the same methodology as for *sorting*. We consider the product of two square matrices of the same size. So, when we say n, it means a product of two n^2 matrices. We use *double precision* data.

In both cases, we do perform the post-processing (*merging* for *sorting* and *matrix addition* for *matrix product*) that is needed to form the final solution. The reason is that this does not provide any information about the ability of our framework to implement parallelism.

Tables 3 and 4 provide the results obtained on a 2 cores machine (Intel Core 2 Duo Processor E8400). In Table 3, n reads $n \times 10^6$.

n	seq	pthread		task		parfor	
	t(s)	t(s)	σ	t(s)	σ	t(s)	σ
1	55	42	1.3	53	1.0	48	1.1
2	120	64	1.9	82	1.5	89	1.3
3	174	97	1.8	125	1.4	123	1.4
4	233	122	1.9	165	1.4	158	1.5
5	300	162	1.8	220	1.4	211	1.4

Table 3. Sorting with 2 cores

n	seq	pthread		task		parfor	
	t(s)	t(s)	σ	t(s)	σ	t(s)	σ
400	2	9	0.2	8	0.2	14	0.1
800	12	16	0.7	18	0.6	20	0.6
1200	43	32	1.3	41	1.0	43	1.0
1600	103	72	1.4	86	1.2	84	1.2
2000	208	135	1.5	166	1.2	158	1.3

Table 4. Matrix-product with 2 cores

We now show the performances on an Intel Core i7-2600 Processor with 4 cores and up to 8 threads (Hyper-Threading). Table 5 (resp. Table 6) provides the speedups with *sorting* (resp. *matrix-product*), and Figure 5 (resp. Figure 6) focuses on the biggest case to illustrate parallelism.

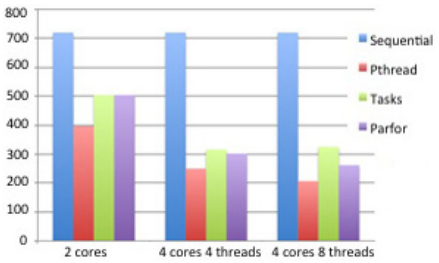

Figure 5. Performance with the *quicksort*

		Pthread	Tasks	Parfor
2 threads	1000000	1,42	1,21	1,13
	2000000	1,63	1,32	1,32
	3000000	1,65	1,33	1,32
	4000000	1,61	1,28	1,28
	5000000	1,82	1,43	1,43
		Pthread	Tasks	Parfor
4 threads	1000000	1,94	1,94	1,79
	2000000	2,37	2,22	2,09
	3000000	2,57	2,14	2,06
	4000000	2,55	2,03	2,14
	5000000	2,88	2,28	2,38
		Pthread	Tasks	Parfor
8 threads	1000000	2,62	1,94	2,27
	2000000	3,11	2,14	2,48
	3000000	3,25	2,17	2,52
	4000000	3,17	2,13	2,41
	5000000	3,48	2,22	2,75

Table 5. Performance of sorting with 4 cores

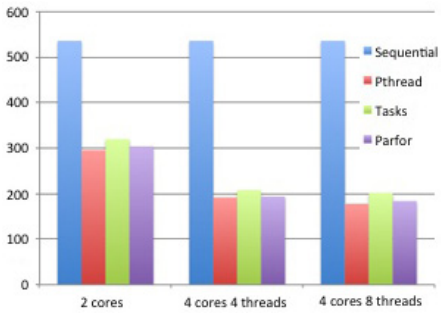

Figure 6. Performance with *matrix-product*

		Pthread	Tasks	Parfor
2 threads	400*400	0,12	0,16	0,12
	800*800	0,74	0,82	0,69
	1200*1200	1,34	1,22	1,25
	1600*1600	1,64	1,52	1,6
	2000*2000	1,81	1,68	1,76
		Pthread	Tasks	Parfor
4 threads	400*400	0,11	0,3	0,19
	800*800	0,89	1,27	1,06
	1200*1200	1,67	1,95	1,8
	1600*1600	2,42	2,42	2,56
	2000*2000	2,79	2,58	2,76
		Pthread	Tasks	Parfor
8 threads	400*400	0,1	0,3	0,36
	800*800	0,99	1,27	1,48
	1200*1200	2,13	1,91	2,18
	1600*1600	3,01	2,42	2,84
	2000*2000	3,01	2,65	2,91

Table 6. Matrix-product with 4 cores

We globally see that parallelism really occurs with a good average speedup. The *Pthread* based solution seems to outperform MATLAB based alternatives regarding *scalability* and *overhead*. Another important advantage using

the *Pthread*-based solution is that we are not limited in the number of threads, thus we may benefit from Hyper-Threading if available. We couldn't do the same with MATLAB bases solution, probably due to some limitations related the number of physical cores or the user profile. Thus, we just double the load of each thread in other to compare with *Pthread*-bases solution in regard to the Hyper-Threading feature.

Figures 7, 8, and 9 show the CPU occupancy rates for each of the parallel solutions, considering the Hyper-Threading feature as previously explained. We see that the occupancy is maximal with the *Pthread*-based solution. With the MATLAB tasks solution, all the (virtual) cores are participating, but under a moderate regime. For the *parfor* based solution, we only have 4 cores participating.

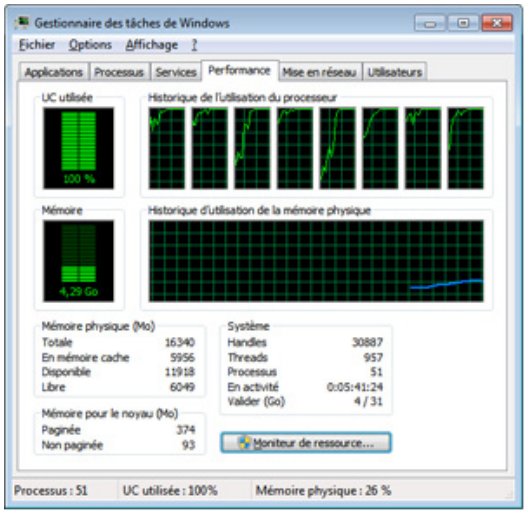

Figure 7. CPU-cores load with Pthreads

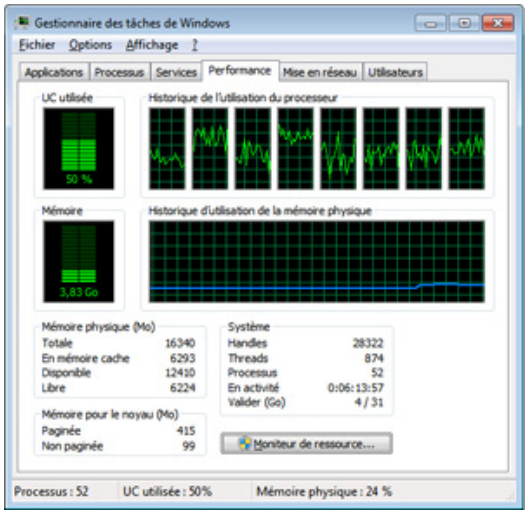

Figure 8. CPU-cores load with MATLAB tasks

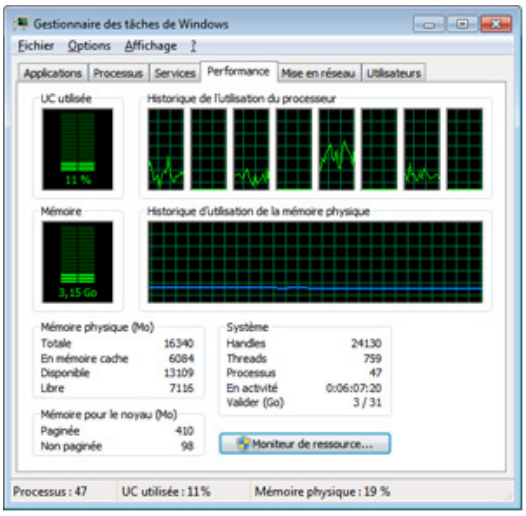

Figure 9. CPU-cores load with *parfor*

6 Conclusion

This paper present our contribution on multicore parallel programming in MATLAB . Our main contribution is based on the *Pthread* library, which is portable standard for threads programming. Connecting MATLAB to this library through a *mex-file*, where each thread launches a MATLAB engine to execute its task, is technically sound. By doing this, the user does not need any additional MATLAB packages to move to parallelism, and our framework provides a quite natural way to request a parallel execution of different MATLAB instructions. Having an intuitive way to express calculations is the main wish of MATLAB users. Experimental results clearly illustrate the effectiveness of our contribution. We think that going this way will boost parallel programming considerations with MATLAB .

Among potential perspectives, we plan to extend the argument of our parallel construct to cover a set of instructions instead of a single instruction, similar to OpenMP sections. The relevant effort is more on the parsing rather than on the heart of the mechanism. Another aspect to study is how to avoid explicit data exchanges between contexts, the solution could be OS dependent because the underlying MATLAB sessions are not always managed the same way. Scalability on systems with larger number of cores should be investigated too.

We plan to make our framework available very soon on the web (code and documentation), likely under the GNU General Public License (GNU GPL).

References

[1] E. Agullo, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, J. Langou, H. Ltaief, P. Luszczek, and A. YarKhan, *PLASMA: Parallel Linear Algebra Software for Multicore Architectures*, Users Guide, http://icl.cs.utk.edu/plasma/, 2012.

[2] J. Burkardt and G. Cliff, *Parallel MATLAB: Parallel For Loops*, http://www.icam.vt.edu/Computing/ vt_2011_parfor.pdf, may 2011.

[3] M. Hill and M. Marty, *Amdahl law in the multicore era*, Computer, vol. 41, no. 7, pp. 33 ?38, 2008.

[4] N. Oberg, B. Ruddell, Marcelo H. Garcia, and P. Kumar, *MATLAB Parallel Computing Toolbox Benchmark for an Embarrassingly Parallel Application* , University of Illinois, http://vtchl.illinois.edu/sites/ hydrolab.dev.engr.illinois.edu/ files/MATLAB_Report.pdf, june 2008.

[5] G. Sharma, J. Martin, *MATLAB: A Language for Parallel Computing*, International Journal of Parallel Programming, Volume 37, Number 1, pages 3-36, February 2009.

[6] C. Tadonki, *High Performance Computing as a Combination of Machines and Methods and Programming*, HDR book, University Paris-Sud Orsay, France, may 2013.

[7] http://www.mathworks.fr/fr/help/matlab/ ref/eval.html

[8] http://www.mathworks.fr/fr/help/matlab/ ref/evalin.html

[9] http://www.mathworks.fr/fr/help/ distcomp/parfor.html

[10] http://www.mathworks.com/help/pdf_doc/ distcomp/distcomp.pdf

[11] http://www.mathworks.fr/fr/help/matlab/ matlab_external/using-matlab-engine.html

[12] http://openmp.org/

[13] http://supertech.csail.mit.edu/cilk/