Position Papers of the Federated Conference on
Computer Science and Information Systems pp. 91–96

DOI: 10.15439/2015F233
ACSIS, Vol. 6

jPar – a simple, free and lightweight tool for parallelizing Matlab calculations on multicores and in clusters

Andrzej Karbowski
NASK, Research and Academic
Computer Network
ul. Wąwozowa 18
02-796 Warszawa, Poland
and
Institute of Control
and Computation Engineering
Warsaw University of Technology
ul. Nowowiejska 15/19
00-665 Warszawa, Poland
E-mail: A.Karbowski@elka.pw.edu.pl

Marek Majchrowski
Warsaw University of Technology
E-mail: M.Majchrowski@coi.pw.edu.pl

Piotr Trojanek
Institute of Control
and Computation Engineering
Warsaw University of Technology
E-mail: P.Trojanek@elka.pw.edu.pl

Tomasz Pokorski
Institute of Control
and Computation Engineering
Warsaw University of Technology
E-mail: T.Pokorski@stud.elka.pw.edu.pl

Dawid Załuga
Institute of Control
and Computation Engineering
Warsaw University of Technology
E-mail: D.Zaluga@stud.elka.pw.edu.pl

***Abstract*—We present a very simple, free tool for parallelizing calculations under Matlab in multicore and cluster environments. After the installation it does not use any compilers, MEX files, disk files, etc. It is compatible with the old Paralize package, but allows the involved cores/machines to do other jobs when a worker core/machine is not busy.**

I. Introduction

THE aim of the described work was to add to the Matlab software a free and lightweight support for distributed and parallel computations. The main idea was to provide a simple, user friendly tool (such as Matlab environment itself), which does not use too much the machines' resources to organize the calculations.

Despite the existence of Matlab Parallel Computing Toolbox (PCT) and Distributed Computing Server (DCS) [1] there is still some sense in developing such tools, because:

- Matlab PCT/DCS are changing, that is, there are some differences between the subsequent versions. For example, in recent versions there were changes concerning two commands: `matlabpool` (which was replaced in R2013b release by `parpool` and completely removed since release R2015a) and `mapreduce`,
- if one wants to perform computations in clusters, except Matlab PCT it is necessary to buy Matlab DCS, which is much more expensive and complicated in administration,
- universities and research laboratories usually buy Matlab licenses in packs; this means, that after hours in labs many licenses are free,
- there is a big interest in free software tools.

Due to these reasons, we observe still a lot of works devoted to developing software environments to parallelize Matlab, both on multicores and in clusters [2], to mention only: *Multicore* [3], *MatlabMPI* [4], *pMatlab* [5] and *shared-matrix* [6].

Almost all free packages for parallelizing Matlab calculations use disk files for communication between Matlab instances, what is inefficient and prone to errors. They are often based on C-MEX files, which have to be ported to every environment (i.e., compiler, operating system), where the tool will be used. Some of them also make use of the remote execution of the child processes through the shells such as `ssh`, `rsh`, which are neither convenient to use, nor available everywhere (e.g., in Windows operating system). Most of these packages introduces plenty of Matlab functions or extend basic syntax to provide communication and synchronization between computing nodes. And last, but not least, these packages do not always work under new versions of Matlab.

Our goal was to find a solution for the described issues and to provide a smart, but easy to install and use tool for parallel and distributed computing, working under different versions of Matlab.

One of the oldest and - no doubts - the simplest software packages to parallelize Matlab calculations is *Paralize* [7]. It consists of only two m-functions: `paralize.m` and `serve.m` of, respectively, 146 and 94 lines (including comment lines) and implements fork-join model of parallel computations, without communication and synchronization between instances. It helps to make calculations both within multiprocessor/multicore machines and in clusters of computers. The biggest drawback of *Paralize* is that communication and synchronization is realized via the disk files, what involves active polling, or, in other words, busy waiting and sometimes causes race conditions errors.

The *jPar* package is as simple as *Paralize* from the user point of view (actually it is compatible with it), very easy to install, but it does not waste the cycles of cores on active polling and allows to use the machines with started Matlab instances to other purposes. Moreover, it does not use for communication and synchronization network filesystem, but more efficient and reliable Java RMI and Java threads.

II. JPAR PROJECT AND IMPLEMENTATION

The basic requirements for *jPar* were:

- to avoid communication via disk files, which is slow and may cause race conditions errors,
- to avoid active polling, that is the waste of time of processors/cores (and energy),
- it should be portable (implemented in Java and .m scripts),
- it should be possibly small,
- it should be easy to install and use.

Distribution of the work in *jPar* is done by dividing the data and executing the same operation on all chunks by slave Matlab processes called *solvers*. The data chunk together with a function to be executed (*task*) is passed to *solvers*. When the calculations are finished, the results are gathered on the *console* (i.e., the *client*) Matlab session (Fig. 1), so the calculations with *jPar* may be interpreted as a simplified implementation of the distributed arrays idea.

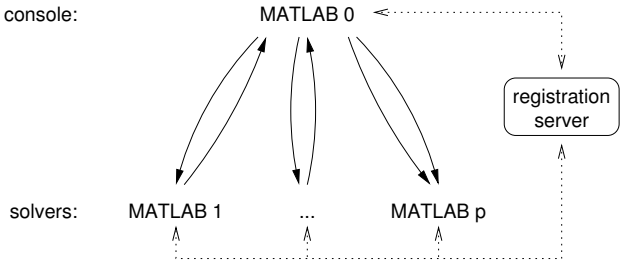

Fig. 1. *jPar* communication model.

The presented approach has been implemented using Java language, owing to its portability and ease of interfacing with Matlab [8]. Synchronization and data exchange between the nodes were done by means of Java *RMI* (Remote Method Invocation) mechanism.

The package consists of only three components:

1) Registration server
2) Solvers
3) Client.

The first process to be run is the *registration server*, which task is to manage the set of *solvers*. It is provided in the form of a single Java executable JAR file (which also contains all the Java classes used by Matlab) and should be started from the command line of the operating system.

The next step is starting several Matlab instances: one for the *jPar console* and the others for *solvers*. The latter should be started from within Matlab sessions (from the same directory) and left. It does not matter what operating systems are used, since both computing (Matlab) and communication (Java) environments are system independent. The registration is done by adding a handle to a remote *solver* object to the managed set. Then the Matlab sessions are blocked until the new tasks are available. The *client* just divides input data into chunks (as in *Paralize*, along the third dimension) and creates partial tasks to perform the distributed job.

In our first version *solver* was implemented as a non-blocking function. Calling Matlab from Java was done by using *JMI* (*Java Matlab Interface*) with functions and classes defined in the Matlab JMI package (JAR file JMI.jar). However, this implementation required running *solvers* within Matlab sessions with GUI display. Another important drawback of this solution was, that the JMI package was distributed by MathWorks "as it is", without any warranty, support or documentation.

Hence, in the final version *solver* is a blocking Matlab function, which waits on a Java object, until it is notified. Then, using provided methods, *solver* gathers all parameters, converts them from Java representation and makes calculations in Matlab. At the end the results are converted and sent back to the *jPar client*.

As it was mentioned before, to perform the computation on multiple nodes, the input data has to be in the form of a three dimensional array. The job is divided into chunks by the partition along the third coordinate, which identifies the task (and the chunk). All the remaining parameters are passed unmodified to *solvers*.

In the example presented below ten *tasks* will be created (each to handle 100×100 matrix):

```
>> a=rand(100,100,10)+i*rand(100,100,10);
>> [V,D]=jpar_client('eig',a);
```

Marshalling and unmarshalling the data objects is done by RMI, but some attention was also paid to the transfer of data between Matlab and Java. While basic types (such as `double`) are automatically converted, the imaginary part of a number is discarded. There is also an issue with vectors, where dimension has to be preserved, since Java does not distinct between horizontal and vertical arrangement of the elements. In these cases Matlab data is converted by the package to the internal representation:

```
public class JMArray
  implements Serializable {

  private Object realpart, imagpart;
  private int dimX, dimY;
  /* ... */
}
```

Since the options of some *solvers* are passed as Matlab structures (e.g., in *fmincon* function from Matlab Optimization Toolbox), the package also converts them to an internal representation:

```
public class JMStruct
```

```
  implements Serializable {

  private Object fields, values;
  private int dimX, dimY;
  /* ... */
}
```

The variables *dimX* and *dimY* are also used to reshape the structure after Matlab→Java→Matlab conversion to match the original arrangement. In *fields* the package sustains the field names of the original Matlab structure, *values* are collected as in the original structure.

It is possible to pass strings or function handles as second and further arguments of *j*Par, that is the actual parameters of parallelized functions. To make it possible, a Java class *JMHandle* in package *matlab.jmhandle* was created. The purpose of this class is to store both the information about the name of the function used to create handle and the absolute path to the file containing that function. When we are passing a function handle to *j*Par, an object - an instance of *JMHandle* class - is created. When the *JMHandle* object is received, a Matlab function handle is recreated basing on the information stored in the object.

It is also possible to pass sparse matrices as arguments. To pass a single sparse matrix as an argument of a function being parallelized no special effort is needed. In order to pass multiple sparse matrices designated for parallelization, a vector of cells containing these sparse matrices must be created and passed as an argument to the function being parallelized. Sparse matrices must be stored in a cell array of size $1 \times N$, where N is the number of matrices. All matrices in this "vector" must be sparse.

For example:

```
>> as=sparse(a);
>> bs=sparse(b);
>> cs=sparse(c);
>> con{1} = as;
>> con{2} = bs;
>> con{3} = cs;
```

When a proper argument is passed, *j*Par converts sparse matrices to vectors that represent coordinates and values. These vectors are passed to *JMSparse* class in Java. After that the data is transferred between the *client* and a *solver*. Before the computations, the *solver* is restoring sparse matrices from the vectors. If results of function being parallelized are sparse matrices, *j*Par puts them into the vector of cells.

The complete implementation of *jPar* has about 800 lines of Java code and 400 lines of Matlab code. It has been tested on computers running both Linux and Windows. It is important to note, that a job can be divided and allocated to *solvers* running under different operating systems and even under different versions of Matlab and Java Virtual Machines.

III. The installation and use of jPar

The *jPar* distribution package containing all the source files may be downloaded from the MatlabCentral Web page [9].

The installation of *jPar* is very simple:

1) To build *jPar* you need to have JDK (Java Development Kit) with `javac` and `jar` tools installed. Start command line (shell) interface and add JDK tools directory to your PATH environment variable if necessary (Windows) / change the proper shell configuration file (e.g.: `.profile`, `.zshrc`, `.cshrc`) (Unix/Linux).
2) Build the `jpar.jar` Java archive with command `"compile.bat"` (Windows) or `"sh compile.sh"` (Unix/Linux).
3) Copy the `.java.policy` file to home directory (in Windows use `"Documents and Settings\Username"` directory) or use `"install.bat"` (Windows) / `"sh ./install.sh"` (Unix/Linux) scripts on every node, where you want to run *jPar client* or *solver*.
4) Check whether the folder with Java binaries is in the system path writing `"java"` in the command window. If it is not, change the path variable, adding this directory (see p. 1).

The installation is performed only once. To use *jPar*, that is to start a session with it, you should:

1) Run one instance of *jPar server* using `"jpar_server.bat"` (Windows) / `"sh ./jpar_server.sh"` (Unix/Linux) on a node, where you want to run *jPar client*.
2) Start Matlab sessions and change the directory to the one which contains *jPar* files (if they were not started from this directory).
3) Start *solvers* from Matlab session in *jPar* directory using:

   ```
   >> jpar_solver(['<hostname>']);
   ```

   where `<hostname>` is the name of host where *jPar server* is running (default to `localhost`).
4) Start a distributed application by the following command:

   ```
   >> [<output>]=jpar_client(...
            '<name_of_the_function>',...
             <parameters>)
   ```

   where both input and output parameters are separated by commas.

After the work is done, the user should kill the *solvers*:

```
>> jpar_client('kill');
```

To see free *solvers* one may use the command:

```
>> jpar_client('hosts');
```

IV. Case study - parallel and distributed optimization

The tests have been performed on a network of Windows PCs with Intel Dual Core 3 GHz processors connected by 100 Mb/s network. All computers had common filesystem provided

by Linux server running *Samba*. The test job was to find the solution of a constrained separable optimization problem:

$$\min_{x\in X}\sum_{i=1}^{p} f_i(x_i) \tag{1}$$

subject to

$$\sum_{i=1}^{p} g_{ji}(x_i) \le M_j, \quad j=1,\ldots,m \tag{2}$$

$$x=(x_1,x_2,\ldots,x_p)\in X=X_1\times X_2\times\ldots X_p \tag{3}$$

$$X_i\subseteq\mathbb{R}^{n_i}, \quad n=\sum_{i=1}^{p} n_i \tag{4}$$

where all functions f_i are strictly convex, g_{ji} - convex, $i=1,\ldots,p; j=1,\ldots,m$.

Big optimization and optimal control problems solved in Matlab environment are often decomposed and parallelized [10]. To solve the above problem in a decomposed, parallel way the classical price method of hierarchical optimization was applied [11]. This method, which is often used to solve network optimization and control problems [12],[13], consists in the decomposition of the minimization of the Lagrangian $L(x,\lambda)$ while calculation of the dual function $L_D(\lambda)$ in the following way:

$$L_D(\lambda)=\min_{x\in X}\left[L(x,\lambda)=\sum_{i=1}^{p} f_i(x_i) + \sum_{j=1}^{m}\lambda_j\left(\sum_{i=1}^{p} g_{ji}(x_i)-M_j\right)\right]=$$

$$=\min_{\substack{x_i\in X_i,\\ i=1,\ldots,p}}\left[\sum_{i=1}^{p}\left(f_i(x_i)+\sum_{j=1}^{m}\lambda_j g_{ji}(x_i)\right)-\sum_{j=1}^{m}\lambda_j M_j\right]=$$

$$=\sum_{i=1}^{p}\min_{x_i\in X_i}\left(f_i(x_i)+\sum_{j=1}^{m}\lambda_j g_{ji}(x_i)\right)-\sum_{j=1}^{m}\lambda_j M_j \tag{5}$$

where λ_j, $j=1,\ldots,m$ are nonnegative Lagrange multipliers. In the natural way we obtain a hierarchical, two-level optimization scheme:

1) Local (slaves') level; the i-th local problem, $i=1,\ldots,p$:

$$\min_{x_i\in X_i}\left[L_i(x_i,\lambda)=f_i(x_i)+\sum_{j=1}^{m}\lambda_j g_{ji}(x_i)\right] \tag{6}$$

2) Coordination (master) level:

$$\max_{\lambda\ge 0}\left[L_D(\lambda)=\sum_{i=1}^{p} L_i(x_i(\lambda),\lambda)-\sum_{j=1}^{m}\lambda_j M_j\right] \tag{7}$$

where $x_i(\lambda)$ is the solution of the i-th local problem (6), $i=1,\ldots,p$.

The algorithm was implemented under Matlab with the hill climbing gradient method on the coordination level and the call of native Matlab `fmincon` solver on the local level. The tests were performed on the Powell20 problem [14]:

$$\min_{y\in R^n} 0.5(y_1^2+y_2^2+\ldots+y_n^2) \tag{8}$$

$$y_{k+1}-y_k\ge -0.5+(-1)^k\cdot k, \quad k=1,...,n-1 \tag{9}$$

$$y_1-y_n\ge n-0.5; \tag{10}$$

To transform this problem to the separable form (1)-(4) the vector x was divided into p parts of the dimension $n_1=n_2=\ldots=n_p=\frac{n}{p}$ (we assumed that $p|n$), what implied the corresponding division of the constraints: the p common ones were treated as global constraints and the Lagrange relaxation was applied to them (the remaining defined the subsequent sets X_i).

Denoting:

$$x_{i_j}=y_{(i-1)n_i+j},\ i=1,\ldots,p;\ j=1,...,n_i, \tag{11}$$

$$x_i=\left[x_{i_1},x_{i_2},\ldots,x_{i_{n_i}}\right]^T \tag{12}$$

$$\left[x_1^T,x_2^T,\ldots,x_p^T\right]^T=y \tag{13}$$

$$f_i(x_i)=0.5\sum_{k=1}^{n_i} x_{i_k}^2, \quad i=1,\ldots,p \tag{14}$$

$$X_i=\Big\{x_i\in\mathbb{R}^{n_i}|x_{i_l}-x_{i_{l+1}}-0.5+(-1)^{k(i,l)}\cdot k(i,l)\le 0, \quad l=1,\ldots,n_i-1\Big\} \tag{15}$$

where $k(i,l)=(i-1)\cdot n_i+l$; $c_j=-0.5+(-1)^{j\cdot n_j}\cdot(j\cdot n_j), j=1,\ldots,m$; $m=p$.

$$g_{ji}(x_i)\begin{cases} 0 & i\notin\{j,mod(j,p)+1\}\\ x_{i_{n_i}} & i=j\\ -x_{i_1} & i=mod(j,p)+1\end{cases} \tag{16}$$

for $i=1,\ldots,p; j=1,\ldots,m$.

$$M_j=-c_j=0.5-(-1)^{j\cdot n_j}\cdot(j\cdot n_j), \quad j=1,\ldots,m \tag{17}$$

we can transform our Powell20 problem (8)-(10) to the separable form (1)-(4).

What concerns the parallelization of the Matlab application code, the only necessary work was to replace the lines:

```
for i=1:p
    [xi(:,1,i),fi(1,1,i)]=pow20_pm_loct(...
        xloc(:,1,i),lambda,ni,p,options_k);
end
```

where `pow20_pm_loct` is the Matlab function solving the local problem (6), with the line:

```
[xi,fi]=jpar_client('pow20_pm_loct',...
                    xloc,lambda,ni,p,options_k);
```

TABLE I
TIMES OF CALCULATIONS WITH *Paralize* AND *jPar* (IN [S]).

n	p						
	1	2		4		6	
		Para-lize	jPar	Para-lize	jPar	Para-lize	jPar
108	1.68	17.7	16.1	12.7	15.8	23.9	21.7
216	14.68	39.5	40.4	33.1	31.5	40.4	41.0
432	58.82	366.8	331.6	100.6	99.5	89.9	84.5
648	2072	2033.2	1832.7	299.7	287.2	192.6	183.5
864	NT	2620.4	2390.5	810.1	786.8	381.3	377.7

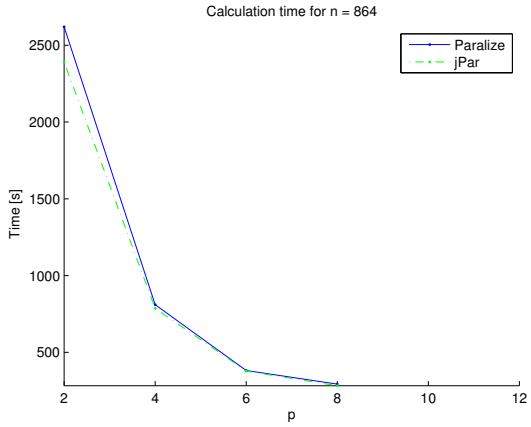

Fig. 2. The speed-up of the decomposition and parallelization for the biggest problem solved

The results are presented in Table I (NT means, that the calculations were not done for such (p, n) parameters) and in Fig. 2.
The superlinear speed-up in some tests was obtained due to a lucky choice of the starting points of optimization for some combinations of problem dimensions and the number of local problems.

Comparing to the original *Paralize*, *jPar* proved to be a bit faster, but more reliable and not "paralyzing" processors of the machines by empty loops.

V. CONCLUSIONS

We presented a very simple, portable package which may be used to parallelize Matlab scripts on multicore/multiprocessor computers with shared and local memory and in heterogenous clusters without a big effort. The only prerequisites are:

- fork-join structure of the application,
- access of all machines to the same file system.

The main advantages of *jPar* are:

- relatively small size,
- simplicity (the time spent on installation and learning it is very short),
- reliability (there are no errors caused by disk transmissions),
- heterogeneity (it was tested on x86 machines under both Linux and Windows),
- interoperability between various Matlab and Java versions,
- not blocking the machines between subsequent chunks of calculations and avoiding flooding of the local network with messages caused by active polling,
- openness, the free use with unlimited number of workers (*solvers*).

JPar is a little ($\sim$ 10%) faster than *Paralize*. Avoiding active polling it does not waste the energy and allows the cores to do other tasks.

Coarse-grained problems, that may be solved in the parallel way, are most suitable for the fast parallelization with *jPar*. The changes in code are very small and their best illustration is the line replacing the `for` loop block presented at the end of the Section IV.

A beta version of *j*Par was already used as the initial environment to perform parallel executions and the communication between master and slaves in MEIGO (MEtaheuristics for systems biology and bIoinformatics Global Optimization) [15] and CeSS (Cooperative enhanced Scatter Search) [16] packages, containing efficient solvers for hard global optimization problems arising in bioinformatics and computational systems biology, based on metaheuristics. This confirms that *jPar* is useful and easy to use not only for computer science specialists.

JPar is free and can be downloaded from the MatlabCentral page [9].

REFERENCES

[1] G. Sharma and J. Martin, "MATLAB®: A Language for Parallel Computing", International Journal of Parallel Programming, vol. 37, 2009, pp. 3–36, http://dx.doi.org/10.1007/s10766-008-0082-5
[2] J. Kepner, "Parallel Matlab for Multicore and Multinode Computers", SIAM Press, 2009, http://dx.doi.org/10.1137/1.9780898718126
[3] M. Buehren, "Multicore - Parallel processing on multiple cores", http://www.mathworks.com/matlabcentral/fileexchange/13775
[4] J. Kepner, "Parallel Programming with MatlabMPI", MIT Lincoln Laboratory, http://www.ll.mit.edu/mission/cybersec/softwaretools/matlabmpi/matlabmpi.html
[5] N. T. Bliss, J. Kepner, "pMatlab: Parallel Matlab Toolbox", MIT Lincoln Laboratory, http://www.ll.mit.edu/mission/cybersec/softwaretools/pmatlab/pmatlab.html
[6] J. Dillon, "sharedmatrix", http://www.mathworks.com/matlabcentral/fileexchange/28572
[7] T. Abrahamsson, "paralize (v2006a)", http://www.mathworks.com/matlabcentral/fileexchange/211
[8] Y.M. Altman, "Undocumented Secrets of MATLAB-Java Programming", CRC Press, 2011.
[9] A. Karbowski et al., "jPar - parallelizing Matlab calculations on multicores and in clusters without file communication", http://www.mathworks.com/matlabcentral/fileexchange/50797
[10] P. Drąg and K. Styczeń, "Parallel Simultaneous Approach for optimal control of DAE systems, in *Proc. Federated Conference on Computer Science and Information Systems (FedCSIS)*, 2012, pp. 587–593.
[11] L.S. Lasdon, "Optimization theory for large systems", Macmillan, 1970; republished by Dover, 2002.
[12] S. Low, and D.E. Lapsley, "Optimization Flow Control, I: Basic Algorithm and Convergence", IEEE/ACM Transactions on Networking, vol. 7, 1999, pp. 861–874, http://dx.doi.org/10.1109/90.811451
[13] A. Karbowski, "Correction to Low and Lapsley's article "Optimization Flow Control, I: Basic Algorithm and Convergence", IEEE/ACM Transactions on Networking, vol. 11, 2003, pp. 338–339, http://dx.doi.org/10.1109/TNET.2003.810318

[14] M. J. D. Powell, “On the quadratic programming algorithm of Goldfarb and Idnani", Mathematical Programmimg Study, vol. 25, 1985, pp. 46–61, http://dx.doi.org/10.1007/BFb0121074

[15] J.A. Egea, D. Henriques, T. Cokelaer, A.F. Villaverde, A. MacNamara, D.-P. Danciu, J.R. Banga and J. Saez-Rodriguez, “MEIGO: an open-source software suite based on metaheuristics for global optimization in systems biology and bioinformatics", BMC Bioinformatics, vol. 15, 2014, http://dx.doi.org/10.1186/1471-2105-15-136

[16] D.R. Penasa, P. González , J.A. Egea, J.R. Banga and R. Doallo, “Parallel Metaheuristics in Computational Biology: An Asynchronous Cooperative Enhanced Scatter Search Method", Procedia Computer Science, vol. 51, 2015, pp. 630–639, http://dx.doi.org/10.1016/j.procs.2015.05.331