# Formation: Matlab: speed up your code

Some files can be downloaded on **http://golgoth.emse.fr/**.

## 1 Introduction to optimization

Matlab interprets (there is no compilation process) the commands (the m-files), and thus is not always very efficient... but you will have to work hard to do better.

Here is one advice that can make your execution go faster :

- Use vectors and operations on vectors as much as you can. For example, prefer this code:

  ```
   x=1:10000;
   y=rand(size(x));
   z=x.*y;
  ```

  to this one:

  ```
   x=1:10000;
   y=rand(size(x));
  tic
   for i = 1:length(x),
       z(i)=x(i)*y(i);
   end
  toc
  exit
  ```

  But this is not the point of this tutorial.

### 1.1 Marty's method

A parallel computation is the use of several processors/machines to compute together the same code. By default, only one processor (one core of processor) is used. Nowadays, even a desktop machine has several cores of processors. This tutorial will show some examples of using the plain capacities of your machines.

This first thing you can think of (for example, when you have a lot a matlab licenses, but no distributed computing toolbox), is to launch several matlab processes on multiple cores or computers. Marty's method is interesting and efficient when you want to apply the same code on different data sets.

This can be done using shell/batch scripts. This tutorial will not explain how to do this.

## 1.2 Compilation

The **École des Mines** owns 1459 licences of the compiler toolbox (as well as 1459 licences of matlab). The compiler can transform your matlab code into an executable. Thus, you dont need any license anymore to execute your program. Moreover, you can execute it on other machines (you have to export the code and the matlab dynamic/shared libraries). The compiler is **mcc**, you call it as you would call **gcc**, the GNU C Compiler. This is not the point of this tutorial.

## 1.3 Parallel computing

This introduces the first real parallelization. Two matlab tooboxes are usefull:

- **Distrib Computing Toolbox**: Allows you to use up to 8 cores on a single machine.
- **MATLAB Distrib Comp Engine**: A (license controlled) number of tokens that you can use on any number of machines. **We are limited to 32 tokens**.

**QUESTION 1.** ***Starter***

☐ Start a web browser and go to **http://golgoth.emse.fr** or **http://clustersms.emse.fr**. Then, go to the ganglia page. You can observe the structure of the clusters.

☐ Start a computer with linux. This is simpler for this tutorial because we need an X server and a ssh client[1]. Start a command window (console or terminal).

☐ Connect to the cluster you want with **ssh -X user@golgoth.emse.fr** or **ssh -X user@clustersms.emse.fr**. Do not forget the option **-X**. **user** is your username (you need an account on one of these machines).

☐ You can start matlab by the command **matlab &**. Notice the character & needed to put the execution into the background. A familiar window should open after a while (the delay is due to the network distance). Beware that this might not be the latest version of matlab.

☐ If you are not familiar with the linux command line, dont worry, you wont need it to much. Just type the following commands:

```
mkdir matlab % creates a directory "make directory"
cd matlab    % go into this directory "change directory"
```

☐ To start matlab without a graphical interface, you can use the command:

```
matlab -nodisplay
```

☐ To ask matlab to start running a m-file maFonction.m (with a function called maFonction), you can type:

[1]Windows users will have to install Xming and putty to get an X server and an ssh client.

```
matlab -r maFonction
```

- ☐ Of course, you can combine the two previous lines:

```
matlab -nodisplay -r maFonction
```

- ☐ PBS is a job manager used to dispatch submitted jobs to the nodes of a cluster. To ask PBS to start a job, here is a shell script you can use (store it into a file called job.pbs):

```
# ask for 20 minutes overall computation
#PBS -l walltime=00:20:00

# cd stands for change directory to...
# a directory called matlab must exist
cd ~/matlab

# start matlab and run the myRand function
matlab -nodisplay -r myRand
```

We are now ready to start some matlab code on a linux cluster.

- ☐ To have an overview on the cluster load, you can type de command

```
pmgcluster -t 2 &
```

## 2 parfor

Usually, parallelizing means **executing the same task several/a lot of times for different parameters**. The **for** loop is the repetition of a task. This first example yields to the use of a parallel for: **parfor**.
**QUESTION 2.** ***Sequential execution***

- ☐ Create a function, called **myRand**, in a file **myRand.m**. It takes one argument **n** and returns the results of **rand(n)**. This function will be called several times.

- ☐ Create a program (m-file) that computes (sequentially with the classical **for**) the random matrices of sizes 20000 to 2007 (8 matrices). You can evaluate the computation time of a command by using **tic** and **toc**. Do not forget to put a “;” after the command, otherwise you will get some huge matrices as a result.

```
>> tic; rand(10000); toc;
Elapsed time is 2.055657 seconds.
```

☐ Modify the file **job.pbs** so that it starts your matlab code. Submit the job to the cluster by the following command:

```
qsub job.pbs
qstat
```

Example:

```
yann@golgoth:~/matlab> qsub job.pbs
34648.golgoth
```

The job manager (PBS) returns the job id. You can check the result of this job by reading the file called `job.o<id>` (replace `<id>` by the job number). Notice that an error file is also created (`job.e<id>`).

```
cat job.o<id> # to read the file
rm job.[oe]*  # to delete all files
```

In these files, you should be able to check the results of the computation.

Well, this is cool, but where is the parallel execution...

**QUESTION 3.** ***parfor loop***

☐ Tell PBS to use 8 cores by modifying the file **job.pbs** and adding:

```
#PBS -l ncpus=8
```

☐ Add the following lines at the beginning of your matlab program.

```
matlabpool open 8
```

Your m-file you look like:

```
x=rand(10000000,1);
y=rand(length(x),1);

% Pre-Allocation of x
z=zeros(size(x));

matlabpool open 8
tic
parfor i=1:length(x)
  z(i)=x(i)*y(i);
end
toc
matlabpool close
exit;
```

WARNING: if you considere the following code, it will run faster than the parallel code. The reason is the pre-allocation of array $z$: matlab will run really fast when arrays are pre-allocated. In fact, if you comment this line, it will compute "for ever"...

```
x=rand(10000000,1);
y=rand(length(x),1);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
z=zeros(size(x)); % pre-allocation !!!
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

t=tic;
for i=1:length(x)
  z(i)=x(i)*y(i);
end
toc(t);
exit;
```

- ☐ Change in your matlab code the **for** instruction by a **parfor**. Submit the job and check the result. The computation time should be improved.
- ☐ You can test different configurations. Read the documentation of the following commands:

```
 doc parfor
 doc matlabpool
```

## 3 Using the "entire" cluster

The parfor command is limited to a single machine (up to 8 cores). The memory is shared by the different threads. To use the maximum of the cluster, another mecanism is available with matlab (equivalent to MPI). It consists in writing **tasks** and creating a job in matlab, and then in asking matlab to submit it to PBS (or to the job manager). Here is the base code of the program. You can download it from the golgoth http server.

### 3.1 Definition of the tasks

```
%% First distributed code with matlab
% This code is the base to introduce the parallel computation with
    matlab.
%
% 1 job is created, containing several tasks. Each task is
   dispatched
```

```
% on the different machines. The myRand.m function is used as an
   illustration.

%% Connection to the job manager
% The cluster uses a job manager (PBS). We must inform matlab to
   use it,
% and also what to tell it.

% jm: job monitor
jm = findResource('scheduler','type','pbspro')

% location of the datas, computed and shared by the tasks.
set(jm, 'DataLocation', '/home/yann/matlab/');

% You can specify that the file system is shared between the nodes
   .
% matlab avoids to replicate all the datas.
set(jm, 'HasSharedFilesystem', true);

% This is the configuration of the CIS cluster golgoth:
% it specifies where to find the matlab executables.
set(jm, 'ClusterMatlabRoot', '/appli/share/matlab');

% Configuration of EACH TASK of the PBS job:
% you can add everything you would have added in a PBS script.
% ncpus: use 1 cpu
% walltime: total time used
% matlabDCE: the most important resource, when you request one
   licence token.
% mem=1gb: request for 1 gb memory per process
set(jm, 'resourcetemplate', '-l ncpus=1,walltime=00:10:00,
   matlabDCE=1');

%% Structure of the Job Manager
% get(jm)

%% Create a job
% A job is constituted of several tasks. Each task will be an
   independant process.
job = jm.createJob;

% Set the name of the job for more fun!
set(job, 'Name', 'toto');

%% Add the local (personnal) path to the different tasks.
```

```
% It is to be preferred to set(job, 'FileDependencies',...)
% because it does not copy all the files over the different nodes
myPath = {'/home/yann/matlab'};
set(job, 'PathDependencies', myPath);

%% Structure of the job
get(job)

% You can get the methods of the job.
% methods(job)

%% Creation of the tasks
%  The tasks are independant processes. Each task (in this case)
% will generate a random square matrix.
% Put whatever you want in these lines.
%
% Warning: the maximum running tasks number depends on the number
   of licences (32).
% Matlab does not deal efficiently with the licences (an error
   occurs), this is why
% PBS is (shoud be) configured to handle them (with matlabDCE
   resource).
for lp=1:35,
    % @myRand : function to be executed
    % 1       : number of arguments of the function
    % {...}   : arguments
  task(lp)=job.createTask(@myRand,1,{lp});
end

%% Submit the job
% Matlab can directly send the job to PBS.
% It is equivalent to the shell command qsub.
submit(job);

% Then, it is the responsability of PBS to start the job.

%% You can wait !
% Now, this script can be stopped. You can exit matlab, take a nap
   , have some coffee
% and come back later to get the results.

exit

% The results will be retrieved in the next program.
```

### 3.2 Get the results

Now that the job has started (you can verify it by using the shell command **qstat**), you can read the next steps: you have to get the results of the computation.

```
% I closed matlab, thus, I have to get back the variables and
   arrays
% as well as the job.

% jm: job monitor
jm = findResource('scheduler','type','pbspro')

% job
j= findJob(jm, 'Name', 'toto');

% Ici, nous allons attendre la fin du job.
% We make sure the job has finished
waitForState(j,'finished');

%% get back all the results
data=getAllOutputArguments(j);
data{1:35}

%% destroy the findJob
% this is mandatory, you have to destroy the job
% otherwise, it will remain for ever
destroy(j);

% end of program
% exit
```

Well, now you are ready to start big parallel programs with matlab. Be carefull: parallel computing does not always need faster computation. You have to take into account the overheads of creating the tasks and submitting the jobs to pbs, and also the network bandwidth and the hard drive writing times, etc...good luck, and remember:
**use the fork, luke !**[2]

## 4 Mex files

If you are an advanced user of matlab, you may have noticed that some matlab executables have strange extensions like **mexa64**... In fact, it means **m**atlab **ex**ecutable on architecture **a**md 64 bits. The general form is `mex<arch>`.

This program is not an m-file that is interpreted. To run faster, it is a compiled code, originally (most of the time) written in C/C++.

[2]pretty good joke, I know

You can create your own mex files and use a powerful library written in C. This is the goal of this section.

## 4.1 Start by reading the matlab documentation

You can start by reading the **Examples of C Source MEX-Files** in the matlab help. It shows you how to start with a basic C code. Edit, understand and compile the **timestwo.c** example of the matlab help. The following code is a modification of timestwo.c. It computes the result on an array instead of on double value.

```
#include "mex.h"

/*
 * timestwo.c - example found in API guide
 *
 * Computational function that takes a scalar and doubles it.
 *
 * This is a MEX-file for MATLAB.
 * Copyright 1984-2007 The MathWorks, Inc.
 *
 * This code has been modified by YG for testing purposes.
 */

/* $Revision: 1.8.6.3 $ */

/*
 * accept arrays
 */
void timestwo(double y[], double x[], int size)
{
    int i;
    for (i=0; i < size; ++i)
    {
        y[i] = 2.0*x[i];
    }
}

/* matlab interface
 */
void mexFunction( int nlhs, mxArray *plhs[],
                  int nrhs, const mxArray *prhs[] )
{
  double *x,*y;
  mwSize mrows,ncols;

  /* Check for proper number of arguments. */
  if(nrhs!=1) {
    mexErrMsgTxt("One input required.");
  } else if(nlhs>1) {
    mexErrMsgTxt("Too many output arguments.");
  }

  /* The input must be a noncomplex scalar double.*/
  mrows = mxGetM(prhs[0]);
  ncols = mxGetN(prhs[0]);
  if( !mxIsDouble(prhs[0]) || mxIsComplex(prhs[0]) ) {
    mexErrMsgTxt("Input must be a noncomplex array of double.");
  }

  /* Create matrix for the return argument. */
  plhs[0] = mxCreateDoubleMatrix(mrows,ncols, mxREAL);

  /* Assign pointers to each input and output. */
  x = mxGetPr(prhs[0]);
  y = mxGetPr(plhs[0]);

  /* Call the timestwo subroutine. */
  timestwo(y,x, mrows);
}
```

Then next subsection will show you a more complex example, with the CGAL library.

## 4.2 Convex hull

The convex hull or convex envelope for a set of points $X$ in a real vector space $V$ is the minimal convex set containing $X$. In computational geometry, a basic problem is finding the convex hull for a given finite nonempty set of points in the plane. It is common to use the term "convex hull" for the boundary of that set, which is a convex polygon, except in the degenerate case that points are collinear. The convex hull is then typically represented by a sequence of the vertices of the line segments forming the boundary of the polygon, ordered along that boundary.

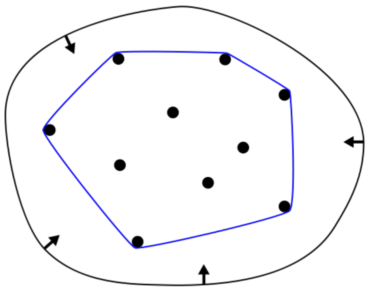

Figure 1: Illustration of the convex hull, from wikipedia.

A really efficient C++ library designed for computational geometry is **CGAL** (developped by INRIA).

#### 4.2.1 C++ code

You can download this code from the golgoth http server. It shows you how to compute the convex hull outside matlab. If the compiled program is called **convhullCGAL_mex.mexa64**, then the matlab command will be:

```
X = randn(1000, 1);
Y = randn(1000, 1);
[x,y]=convhullCGAL_mex(X, Y);
plot(x,y);
```

This following code is compiled with the command line if CGAL is installed in `/appli` (golgoth configuration). You can call this line either from matlab or from the linux shell command line.

```
mex -I/appli/include -L/appli/lib -lCGAL convhullCGAL_mex.cpp
```

The C++ code is:

```
/// This illustrates the use of an extern library (written in C++)
/// within matlab. A executable file is generated (file mex<arch>)
    and called with
/// a particular syntax
///
/// Compilation is called by (not necessarily within matlab):
///
/// mex -I/appli/include -L/appli/lib -lCGAL convhullCGAL_mex.cpp
///

// matlab instructions
#include "mex.h"
extern void _main();

// STL c++
#include <list>
#include <iterator>
#include <iostream>

// CGAL must be installed
// include files for CGAL
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/ch_graham_andrew.h>

typedef CGAL::Exact_predicates_inexact_constructions_kernel K;
typedef K::Point_2 Point_2;
```

```
/// Reads n points in tables X and Y, and insert them into
/// the points structure.
///
/// Lecture de n points dans les tableaux X et Y, et insertion
/// dans la structure points
/// \param n nombre de points
/// \param X tableau des abscisses
/// \param Y tableau des ordonn es
/// \param points liste de points    la mode STL c++
void matlab_input(int n, double *X, double *Y, std::list<Point_2>
   &points)
{
  for (int i=0; i<n; ++i)
    {
      points.push_back(Point_2(X[i], Y[i]));
    }

}

/// Writes the list of points into matlab datas.
/// \param points liste of points
/// \param x X coordinates to write
/// \param y Y coordinates to write
void matlab_output(std::list<Point_2> &points,double* x,double* y)
{
  // iterateur sur la liste de points
  std::list<Point_2>::const_iterator it = points.begin();
  int compteurIt=0;
  while(it != points.end())
    {
      x[compteurIt]=(*it).x();
      y[compteurIt]=(*it).y();
      ++compteurIt;
      ++it;
    }
}

/// Link with matlab
/// When this function is called, matlab defines the pointers and
   datas:
/// for ex: [a,b,c,...] = fun(d,e,f,...)
///
///
/// Fonction qui permet le lien avec matlab
```

```
///
/// Quand la fonction est invoqu e , matlab renseigne lui  m me
   les pointeurs et donn es:
/// p. ex: [a,b,c,...] = fun(d,e,f,...)
///
/// \param nlhs number of ouput tables
/// \param plhs output tables
/// \param nrhs number of input tables
/// \param prhs input tables
void mexFunction(int nlhs, mxArray *plhs[],
                 int nrhs, const mxArray *prhs[])
{
  /* Check for proper number of arguments */
  if (nrhs != 2)
    {
      mexErrMsgTxt("MEXCPP requires two input arguments.");
    }

  // number of points for the convex hull
  int nElts = mxGetM(prhs[0]);

  // X coordinates
  double *X = (double *) mxGetPr(prhs[0]);

  // Y coordinates
  double *Y = (double *) mxGetPr(prhs[1]);

  // structures for the coordinates, to be passed to CGAL
  std::list<Point_2> points;
  std::list<Point_2> resPoints;

  // Creation of the list of points
  matlab_input(nElts, X, Y, points);

  // Computation of the convex hull.
  // the result is stored into resPoints
  CGAL::ch_graham_andrew( points.begin(), points.end(),std::
     back_inserter(resPoints));

  // Results: we create 2 tables with the right size.
  plhs[0] = mxCreateDoubleMatrix(resPoints.size(), 1, mxREAL); //
     mxReal is our data-type
  plhs[1] = mxCreateDoubleMatrix(resPoints.size(), 1, mxREAL); //
     mxReal is our data-type
  double* x = mxGetPr(plhs[0]);
  double* y = mxGetPr(plhs[1]);

  // On  crit les points dans les tableaux matlab, ce qui permet
     de les r cup rer
  // The results are written into the output tables.
  matlab_output(resPoints,x,y);
}
```

## The End

If you have any comment, please send me an email at gavet@emse.fr.