The HUB2CAC Multi-target MATLAB Framework

by Pascal Meunier, Steve Clark and Brandon Hill

Document Version 0.12, Nov 29, 2010

Table of Contents

1. Scope and Motivation

The goal of this framework is to facilitate the adaptation and management of existing MATLAB code for multiple targets, or creating new code intended to run on those targets. The targets can be specific clusters, e.g., the Center for Advanced Computing (CAC) at Cornell University (http://www.cac.cornell.edu/), generic MATLAB clusters, or hubs, e.g., nanoHUB (http://nanohub.org/), and local resources. The management of the code is done in a manner respecting the MATLAB licensing agreements.

This document describes what can be done with the framework (use cases), and how to use it. It is addressed to people who are considering whether to use it, as well as MATLAB code developers whose code may now or later require additional computing resources.

2. Technologies Used

The HUB2CAC framework uses MATLAB scripts, Java code, Python scripts, and shell scripts. These are managed with Makefiles (for the "make" program) specifying how to generate code depending on the desired use. It has been tested only for development on Linux platforms, although the deployment targets may be Windows clusters.

3. Use Cases

3.1 Local Scheduler

Adam is debugging his code and wants to see if it runs correctly, without the overhead of sitting in a queue and obtaining credentials for the use of a cluster . He types "make localInstall" from the "src" directory. This generates a "local_sched" file which can be run directly by the local installation of MATLAB:

local_sched <number of tasks>

It separates the execution into multiple parts that are run as if they were a cluster. The results from the separate runs are then recombined to produce the final result by a second script that was generated at the same time (with a name he chose). When he's done he can run "make localclean" or "make localDistclean" to remove unneeded files.

3.2 Direct CAC Runs

Barbara has her own credentials (certificate) to run MATLAB jobs at CAC (Center for Advanced Computing at Cornell University). She types "make CACinstall", which compiles the MATLAB code and transfers it to CAC in a directory it creates based on the tool's name. Thereafter, she can invoke the code at CAC with different sets of data with:

<toolname>CAC.py <CAC user name> <number of tasks>

However, Barbara's program is complex and requires customized data for each task. She puts the appropriate data in <number of tasks> folders before invoking the above script. All her data is automatically transfered to CAC. Her program also produces output in the form of files. The files she wants are transfered back to her workstation and the results are assembled by another script:

```
run_<Barbara's post-processing script>.sh  <number of tasks>
```

When Barbara doesn't need her program anymore, she types “make CACclean” or “make CACdistclean”. This also deletes the code in her CAC account and the associated directories, so they don't count against her quota.

3.3 HUB Tool Development and Private Runs

Charles is a graduate student planning to publish his tool once his paper has been accepted for publication. He has an account on a gateway (hub) which allows him to use clusters through the gateway's credentials. When he changes his code, he types “make devInstall”. This compiles his code and transports it to CAC, using the gateway's subsystems and accounts.. When he wants to run the tool, he has the choice of using a graphical interface (Rappture in the case of nanoHUB) or running it from the command line. In the later case, he types:

```
<toolname>.submit  <number of tasks>
```

His raw data or tables, if any, are automatically transfered to CAC, and the results retrieved, by the hub's systems.

3.4 Publishing a HUB Tool

Diana manages the publication of tools on a gateway. Charles submitted his code for publication; Diana types “make prodInstall”. This generates code for use by the special system account in the gateway that runs tools on behalf of users, and transfers appropriate parts to CAC. Diana puts the other generated code in the appropriate area of the system and updates the gateway to make the tool available to the community.

Whenever interactive users request simulations, the framework will automatically select an appropriate queue at CAC which is likely to yield a low latency and good performance.

3.5 Generic Portable Code

Eric wants to run his MATLAB code on a different, generic cluster. The MATLAB license requires him to compile the code, and besides he doesn't want others to be able to see the source. He types “make portInstall”, which generates code he can transfer by sftp or otherwise.

This use case also applies to gateways who determine that another cluster than CAC needs to be used, perhaps because CAC is very busy at the moment.

4. Internal Framework Organization

The framework uses the following directories: bin, data, examples, run_dir, and src. An important concept is that each task runs is a separate directory. All files in a task's directory are automatically transfered to clusters; so it is sufficient to copy files in the directories <run directory>/part1, part2, part3, ... partX before submitting a job, for them to be transfered. This allows a tool author to specify complex and different run parameters for each task, if so desired. The bin directory contains the compiled code and scripts. The data directory contains resources that will be used in common by all the tasks. The examples directory contains the factorial and pi-rectangle example tools, discussed in the next section. The run_dir directory, used for testing convenience, is not to be confused with the actual run directory when a tool is deployed. The directory "run_dir" is used as a convenience for development and command-line runs, as well as for testing. Gateways are expected to create uniquely named directories for each of the simulations they do on behalf of users. Within the run directory, subdirectories should be created for each task. This is where tasks will read data and put results. There are no restrictions on the location of run directories. The src directory is where the make commands should be typed. The code for a tool should go in a directory named "src/tool".

5. Tutorials by Example

The two examples described below utilize different ways of passing results. The factorial example returns a string, whereas pi-rectangle writes to a file instead. Also, the factorial example gets as an argument the name of a file to be read. These demonstrate how a different number of arguments can be passed to your program, and that result retrieval is flexible.

5.1. Factorial

To start writing a tool, go into the src directory and create the "src/tool" directory, if it doesn't exist already (first delete it if it's a symlink). Normally you would put in it any already existing code there. Copy the files tool_constants.mk and tool_Makefile from src to tool. Rename tool_Makefile to Makefile. Then open the tool_constants.mk file, and set the name of the tool:

```
TOOLNAME = factorial
```

This will tell the framework that generated files and directories should be based on the "factorial" name. We will put our code in a file that we decide to name "fact.m". We update the tool constants:

```
MAIN_FILE = fact.m
```

To calculate the factorial of a number N, we will write N in a file, and read that file in fact.m. An important concept is that each task runs in a separate directory. All files in a task's directory are automatically transfered to clusters; so it is sufficient to copy the file in the directories <run directory>/part1, part2, part3, ... partX. This allows a tool author to specify complex and different run parameters for each task, if so desired.

In each task, we will multiply M numbers, where M is N divided by the number of tasks. At the end, we will multiply the partial products together to get the factorial. Each task needs to know which part of the calculation it needs to perform. We will figure it out from the task ID, the total number of tasks, and the name of the file we should read for supporting data (let's call it "test_part.txt").

```
MAIN_ARGUMENTS = taskId,nTasks,'test_part.txt'
```

The fact.m script (full code listing available in the appendix) reads the above values, does the multiplication and prints out the result:

```
function [result] = fact(taskId, nTasks, number_file)
...
for n=min:max
   fact = fact * n;
end
result = sprintf('%f', fact)
```

The framework will harvest the returned values ("result" in this case) from all the tasks and transfer them back. In the appendix code, you may notice that the input values are checked. If they are of type string, we convert them to integers. This is because their type changes depending on whether the function is invoked from the command line, or by another MATLAB script. To communicate an error condition, for example if we can't open the specified file, we return a string instead of a number.
The final multiplication of the partial products will be done locally by another script, which we called "gather_parts.m". So in tool_constants.mk, set:

POST = gather_parts.m

The post-processing scripts are passed the number of tasks as an argument. From that, the script must read the results from each task directory ("partX"):

```
results_fname = sprintf('part%d/results', i1);
resultsFid = fopen(results_fname);
```

For example, if you look in the part1 directory after a run, you'll find a file named "results" with a single number in it. This is not the only way to return results; the pi-rectangle example uses other files. The only requirement on the generated final results is when Rappture is used, so that they can be rendered and displayed to the user. See the Rappture documentation for details.

5.2. Pi-Rectangle Calculation

We will use the pi calculation method described at http://mb-soft.com/public3/pi.html and figure out the calculations to be performed simply from the task number and number of tasks. As in the factorial example, create the tool directory and copy the files tool_constants.mk and tool_Makefile. Edit tool_constants.mk to specify:

TOOLNAME = pi_rectangle

MAIN_FILE = main_pi_rectangle.m

MAIN_ARGUMENTS = taskId,nTasks

FILES_TO_RETRIEVE = 'psum.mat'

POST = gather_parts.m

We will write results in psum.mat. Our program is composed of more than one MATLAB script. The main script reads configuration parameters from a file named 'pi_rectangle.dat' and then calls another MATLAB script that does the actual computation. An important concept is that each task runs in a separate directory. All files in a task's directory are automatically transfered to clusters; so it is sufficient to copy the file 'pi_rectangle.dat' in the directories <run directory>/part1, part2, part3, ... partX. This allows a tool author to specify complex and different run parameters for each task, if so desired.

The script “pi_rectangle.m” is where the actual calculations take place. It prints messages; you can see the output after the run by looking at the file named <toolname>_output.txt, in this case pi_rectangle_output.txt. It contains the messages printed to standard output from all tasks. Other files

5.3. Other Mechanisms

5.3.1. Constant Data Directory

If a tool has data that remains constant for all invocations of the tool, it can be placed in the “data” directory (at the same level as src). When an install command is given, the data is transfered to the cluster and is not transfered again when the tool is invoked.

5.3.2. Helper Binaries

Some tools may need to call C programs. To cater to these special cases, the tool Makefile needs to be customized. This is an area that will be improved in the next framework release.

5.3.2. Submit mechanism

The mechanism that accepts jobs and runs them on behalf of gateway users is called “submit”. It is published as open source in the HUBzero gateway software.

6. Appendix: Code Listings

6.1. Factorial Example

6.1.1 Fact.m

```
function [result] = fact(taskId, nTasks, number_file)
   % taskId between 1 and nTasks
   if isstr(taskId)

      taskId = str2num(taskId);

   end
   if isstr(nTasks)
      nTasks = str2num(nTasks);
   end
   fid = fopen(number_file);
   if fid == -1
      result = number_file;
      return
   end
   N = fscanf(fid, '%d')
   fclose(fid);
   span = N/nTasks;
   if span < 1
      result = '0'
      return
   end
   max = taskId * span;
   min = round(max - span) + 1
   max = round(max)
   fact = 1;
   for n=min:max
      fact = fact * n;
   end
   result = sprintf('%f', fact)
   return
end
```

6.1.2. Gather_Parts.m

```
function gather_parts(N)
   factprod = 1;
   if isstr(N)
      nTasks = str2num(N);
   else
```

```
      nTasks = N;
   end
   for i1=1:nTasks
      results_fname = sprintf('part%d/results', i1);
      resultsFid = fopen(results_fname);
      if resultsFid == -1
         fprintf('Error: results file not found!\n')
         return
      end
      partfact = fscanf(resultsFid, '%f');
      factprod = factprod * partfact;
      fclose(resultsFid);
      end
   fprintf('The factorial is %f\n', factprod)
end
```

6.2. Pi_Rectangle Example

6.2.1 Main_Pi_Rectangle.m

```
function [result] = main_pi_rectangle(processor,nProcessors)
result = 0;
if isstr(processor)
   processor = str2num(processor);
end
if isstr(nProcessors)
   nProcessors = str2num(nProcessors);
end

fp = fopen('pi_rectangle.dat','rt');
j1 = 0;
while 1
   value = fgetl(fp);
   if ~ischar(value), break, end
   j1 = j1+1;
   nIntervals(j1) = str2num(value);
end
fclose(fp);

integralValue = pi_rectangle(processor,nProcessors,nIntervals);

save psum.mat nIntervals integralValue;
```

6.2.2. Pi_Rectangle.m

```
function [integralValue] =
pi_rectangle(processor,nProcessors,nIntervals)

% calculate PI - method described at http://mb-
soft.com/public3/pi.html

h = 1./nIntervals;
partialSum = zeros(size(nIntervals));
for j1=1:length(nIntervals)
   for i1=processor:nProcessors:nIntervals(j1)-1
      x = h(j1)*(i1+0.5);
      partialSum(j1) = partialSum(j1) + f(x);
   end
end
integralValue = h.*partialSum;

fprintf('Partial sums for processor %d:\n',processor);
fprintf(' %15d   %g\n',[ nIntervals ; integralValue ]);
```

6.2.3. Gather_Parts.m

```
function gather_parts(nP)
if isstr(nP)
   nParts = str2num(nP);
else
   nParts = nP;
end
```

```
for i1=1:nParts
   pSumFile = [ pwd '/part' num2str(i1) '/psum.mat' ];
   load(pSumFile);
   if i1 == 1
      piSum = integralValue;
   else
      piSum = piSum + integralValue;
   end
end

estimateError = abs(piSum - pi);
fprintf('\n      Intervals    Pi estimate      Error\n');
fprintf('%15d   %12.10f   %12.10e\n', [ nIntervals ; piSum ;
estimateError ]);

fp = fopen('pi_rectangle.out','wt');
fprintf(fp,'%d %g\n',[ nIntervals ; estimateError ]);
fclose(fp);
```

7. Code Changes and Bug Reports

If you would make improvements to the framework and would like to communicate them back to us, we suggest using the git version control system. The easiest way for us to look at your code is if you checkout a copy of the code from our git repository. Then, publish your changes from your own repository; we'll fetch the changes and examine them. You can tell us about the changes by opening a ticket on HUBzero or nanoHUB, or by emailing the authors.

8. Acknowledgements

This work was funded by NSF grant #0844032, "A TeraGrid MATLAB Cluster - Exploring New Services for an XD Future" (http://nsf.gov/awardsearch/showAward.do?AwardNumber=0844032)