Automatic task-farming.

M. Oliveira[1,2]

[1] Laboratório de Instrumentação e Física Experimental de Partículas, Pólo de Coimbra
[2] Centro de Física Computacional and Departamento de Física
Universidade de Coimbra

Abstract. HPC systems always have special policies to increase productivity of large capability applications. This will invariably hinder attempts by any user to achieve high throughput on any serial code. Grid system always have a time overhead on job submission that render bulk submission impractical. In this paper we describe a very simple MPI application, based on the task farming concept, to automatically deploy any list of serial single processor tasks on these environments.

Keywords: HPC, GRID, MPI, Task farming.

1 Introduction

HPC systems are always configured to increase productivity of large capability applications. In some cases the granularity of the number of nodes allowed to be requested even blocks serial codes automatically. However there are a number of cases where single processor codes are not to be excluded: parameter sweeping applications, hard or impossible to paralelise algorithms, untrained pool of users for a given HPC center, etc.
On Grid systems there is always a time overhead that render bulk submission impractical.
In these situations it would desirable if these systems could give the users an automatic tool to deploy in high throughput a list of serial one processor tasks.
The Milipeia cluster, the HPC resource at the Advanced Computing Laboratory in the University of Coimbra, Portugal, has long felt the need for such an application. This paper details the solution developed by us, as a very simple MPI code, for this situation.

2 MPI and the task farming model.

MPI is an abstration layer based on the concept of communication between processes. When an MPI code is run we must specify the number of processes to be run and/or the list of computers where they will be run. These processes will then communicate by exchanging messages.
In the task farming model one of these processes, the master, is responsible for generating a list of tasks and all the other, the slaves, for actually running them. The messages exchanged between master and slaves occur hence only at most in two instances: at the beginning, where only the task is communicated to the slaves, and eventually at the end, where the slaves may return the result of the computation.
In our particular case the list of tasks is provided by the user and the master only needs to read, interpret it and communicate to each slave its subset of tasks. Moreover, since our master has a very limited set of chores to do, we retain a subset of tasks to execute as well, *i.e.*, after reading and distributing the tasks the master becomes a slave.
After reading the tasks the code also verifies if the number of tasks given is divisible by the number of cores the user requested stopping if the premise is false. This is a minimal attempt at ensuring that all cores are evenly charged during compute time. It is however left to the responsibility of the user to deploy tasks that are similar in terms of time consumption.
The task list must just be any valid underlying shell command, commands or scripts, as all tasks will be given to the computer by a call to the `system()` function.
We should also stress that the usual task farming model requires all tasks to be independent. This might not be globally true in our case. Each group can depend on the preceding. It is however usually not advisable to do so and scripting might become too intricate to sustain synchronisation.

3 The task farming code

Our task farming code has been implemented in C. This is available in all HPC and Grid resources and is also supported by all MPI implementations.
The code has three main parts: reading the task list, only executed on the master, communicating the sub-tasks and executing each one, where master and slaves all participate.
For reading the task list we decided to use a dynamic structure with a pointer to an array of pointers `**char`. Every character pointer `*char` substructure will correspond to a task. Each character is read in turn from file and appended to a pre-existing pointer `*char` substructure or to a newly allocated one depending on whether it is preceded by a new line character or not. Empty and commented lines are disregarded. As a safeguard the list of tasks read from the user's file is then written to an auxiliary file so that each user can check if something was not read as intended.
Reading the task list with a dynamic structure in this way we impose no other limits on the code or on the task list than system limits.
As explained previously after reading the tasks the code also verifies if the number of tasks is divisible by the number of cores the user requested and will stop if the premise is false.
After this stage the code enters the communication part where only the tasks required by a single process are exchanged thus minimising data communication. A single `MPI_Bcast` is used to inform the slaves of how many tasks they will receive allowing them to construct the `**char` replica data structure accordingly. The code uses then a double pair of `MPI_Send` and `MPI_Recv` to transmit the length of the task, used to construct the `*char` substructure, and then the task itself.
Finally each process, master and slaves, will execute each task in turn with a `system()` call:

```
for( itasks=0 ; itasks < ntasksperproc ; ++itasks )
     system(*(tasks+itasks));
```

4 Task farming serial applications

Using our task farming application is trivially simple to write a PBS script to submit a job to run multiple serial one processor applications. On our HPC cluster this will be a script like:

```
#!/bin/csh
#PBS -A my_project
#PBS -S /bin/csh
#PBS -l walltime=00:10:00
#PBS -l nodes=4:ppn=4

module load comp/mpich2/pathscale3.0
module load taskfarm

cd $PBS_O_WORKDIR

mpipbsexec taskfarm ./tasks
```

where `./tasks` is just a file on the submission directory with a list of tasks:

```
# My first task:
my_exec_1 < my_input_file_1 > my_output_file_1
# My second task:
my_exec_2 < my_input_file_2 > my_output_file_2
...
# My n-th task:
my_exec_n < my_input_file_n > my_output_file_n
```

As previously stated the number of tasks in this file must be an integer multiple of the requested number of cores. In the example given the number of tasks could only have been 16, 32, 48, 64, etc.

5 Task farming OpenMP applications

Although the task farming application is better suited to deploy serial codes it can also be used to run any other application that is not an MPI code by itself. This open the possibility of running OpenMP applications. If a given system has multi-core capabilities on each node then this even becomes a desirable possibility.
On the Milipeia cluster we have nodes with 4 available cores so a task file to deploy an OpenMP code might look like:

```
# Job for the first node
export OMP_NUM_THREADS=4 ; ./openmp_code
/bin/hostname
/bin/hostname
/bin/hostname
# Job for the second node
export OMP_NUM_THREADS=4 ; ./openmp_code
/bin/hostname
/bin/hostname
/bin/hostname
...
```

Each node is charged with a proper job and three phantom `/bin/hostname` tasks.

6 Task farming on Grid environments

Most grid middlewares now provide support for MPI jobs. The current EGEE Grid middleware supports the "MPICH" job type which is sufficient to deploy our task farming application. The Int.Eu.Grid project allows extra functionality that might allow users extra freedom but which is not required for simple use of the task farming code.
A simple jdl file to deploy this model is:

```
Type          = "job";
JobType       = "Parallel";
SubJobType    = "mpich";
NodeNumber    = 4;
Executable    = "taskfarm";
Arguments     = "tasks";
StdOutput     = "taskfarm.out";
StdError      = "taskfarm.err";
InputSandbox  = {"taskfarm","tasks"};
OutputSandbox = {"taskfarm.out","taskfarm.err"};
```

This would allow a user to quickly deploy a list of tasks on a single submission command.

7 Conclusions

A very simple task farming application was presented here to allow users to deploy serial code in high throughput.
On our HPC system this very simple resource has allowed a vast number of users to take full advantage of a new computational resource with very little or no investment on their part.