Kiko>
A personal job scheduler

V1.4

Carlos allende prieto

june 2022

kiko> is a light-weight tool to manage non-interactive tasks on personal computers. It can improve your system's throughput significantly by making sure the system is never idle, starting new jobs as soon as resources are freed up by previously completed tasks.

kiko> is written in Perl, by default available on any modern unix-like OS (e.g. linux, Solaris, Mac OS). It requires no additional libraries, making installation a trivial process needing no special privileges. A simple, minimalist, command-line interface allows the user to add more tasks to the queue, remove them, and track their progress.

1. Introduction

For practical reasons, large computations are commonly split in smaller sections or jobs. In many cases, these jobs are completely independent, making them suitable for a straightforward parallelization. A typical example is when we need to process many data sets with the same algorithm, or when running Monte-Carlo simulations.

A variety of task scheduling software targeted to massive distributed computers, or large pools of independent small systems, is available (e.g. Condor, PBS, Hadoop, SGE). This is a extreme case in which large numbers of users and jobs with vastly different sizes in terms of length and memory usage share the same resources.

Another extreme situation involves the ubiquitous personal computers. These days, chances are that your desktop has somewhere between 2 and 8 cores. Even laptops have now usually 2 cores, and the race is on for higher multitasking capabilities. One can take advantage of multi-core personal computers to increase computing throughput by running many jobs simultaneously. A job scheduler can be used to help submitting the jobs in an orderly manner, making sure that the system is never idle, and that its resources are not overwhelmed.

While tools intended for use on massively parallel multi-user and multi-node systems are usually capable of handling a one-user few-cores desktop computer, they are cumbersome to install and use.

Kiko> aims precisely to fill this niche, by providing an extremely light-weight tool, with essentially no installing requirements, able to satisfy most users' needs to run many non-interactive tasks on their own machines.

2. Requirements

-A unix-like OS (that includes, of course, linux, Solaris, and MacOS X)
-Perl 5 (typically a default if your system satisfies the condition above)

3. Install

Download kiko> from http://github.com/callendeprieto/kiko. Kiko> is a perl program and the only thing needed to run it, besides Perl, is the source code, which is all contained in a single file named, you guessed it, **kiko**.

Make sure that kiko has permission for execution ("chmod u+x kiko"). It must be in your path to be accessible from anywhere, and so should be the current directory, ".", for kiko to function properly. If you are using bash and you copy kiko to a folder "bin" in your home directory, this can be done by including in your $HOME/.bashrc file

export PATH=$PATH:$HOME/bin:.

Kiko> uses a directory in your filesystem to store information about the jobs that is handling. By default this directory is named '.kiko' and placed in the folder defined by the environmental variable WORK, or if that's missing, in your HOME directory. Installing kiko> involves creating this directory, if it doesn't exist, which can be accomplished by typing:

$kiko --setup

4. How it works

4.1. starting/stopping the deamon

Kiko> is a deamon, i.e. a program that is running in the background. Users submit jobs to the queue using another instance of kiko> himself, and the deamon keeps an eye on the computer's load and

starts a new job when this drops below a threshold. To start up kiko> type

$kiko --on

and it can be similarly stopped

$kiko --off

4.2. submitting jobs

There are two basic pieces of information that kiko> needs to know about your job to be able to run it : the executable file (the job itself, probably a script) and the location (directory) where you want it to be executed. By default, kiko assumes that the directory to execute the job is the same given as the path to the job. If the executable is in your default path, so that you don't explicitly give a path for it when calling kiko>, then the default directory is where you submit the job (pwd). Thus, to submit a job to run the job exe in your current directory, you would usually type

$kiko exe

If you want to specify a directory different from the current one, you would type

$kiko –dir exe

To keep things simple, exe must be an executable file residing at, or visible from, the executing directory, or a one-word command. If your job consists of a longer command, or a sequence of commands or executables, you need to place them in a single file, and make it executable (e.g. using 'chmod').

Let's see a few examples. If you would like to execute the command 'df –h' in the current directory, you would create a file, say a bash script named job1.bash, like this

```
#/bin/bash
du –h
```

and, making sure you have permissions to execute it, submit it to the queue typing

$kiko job1.bash

If you have a bunch of jobs named task1.job, task2.job, task3.job, etc. to be launched from the current directory, you can place them on the queue typing

$kiko *.job

If you have ten identical jobs, say a Cshell script named ‘task.csh’, that you want to run repeatedly in each of the local directores tmp0,tmp2, …tmp9, you would type

$kiko tmp?/task.csh

If you wish to execute the program ‘myprogram’, which corresponds to an executable placed somewhere in your path, twice: one from the local directory ‘dir1’ and and another from ‘dir2’ you would type

$kiko –dir1 myprogram –dir2 myprogram

Anytime you add a –dir to your call, this changes the default executing directory to dir. Therefore to run ‘myprogram’ from the directory ‘d1’ and then ‘myprogram2’ and ‘myprogram3’ from the directory ‘d2’ you would use

$kiko –d1 myprogram –d2 myprogram2 myprogram3

Kiko> will redirect your standard output and standard error output to files in the executing directory **i**.out and **i**.err, where **i** will be the job’s unique identifier (jobid), which will be assigned by kiko> when you submit the job.

4.3. checking job status

This is as easy as typing

$kiko

4.4. removing jobs

When kiko> is called with no arguments, as described in the previous section, it will report back which jobs are *running*, and which ones are *pending* and therefore waiting in the queue for an available slot. For the jobs that are running, kiko> reports their *jobid* and their *pid* – the process id assigned by the operative system. You can use the unix command kill (-9) to kill any job(s) currently running using the *pid*, or ask kiko> to do it for you, by passing it the (usually smaller) *jobid*

$kiko --kill *jobid(s)*

Jobs that are pending (waiting in the queue) have not yet a (OS-based) *pid* but they already have a kiko> *jobid*. To delete one or several use

$kiko –-del *jobid(s)*

There are short cuts to wipe clean the list of pending jobs

$kiko –-delall

as well as all running jobs

$kiko –-killall

NOTE: child processes spawn from a job will not be killed by ‘kill’ or ‘killall’, only the parent job submitted by kiko will.

4.5. configuration options

Kiko> needs to know how many threads your system can handle. This, and other options, can be passed on to kiko> using the configuration file ‘kiko.config’, which lives in your home **.kiko** directory (see Section 3). The default is to use all available threads. To change this, edit your kiko.config file and set the variable **$ncpu** to a different value.

For example, if your system has six 16 but you want to use only 4, edit ~/.kiko/kiko.config (or $WORK/.kiko/kiko.config if you have defined the WORK environmental variable) and write in it

$ncpu=4;

The syntax is pure-Perl, and therefore more complex statements are possible. One could, for example, set the number of available threads to 4 from midnight to 8am, and to 2 for the rest of the time by writing in the config. file

```
($hours,$day,$month,$year)=(localtime)[2,3,4,5];
if ($hours<8){$ncpu=4}else{$ncpu=2};
```

By default kiko> evaluates whether a new process should be launched every 30 seconds, but this can be easily changed using the configuration file. For example, to have checks every 10 seconds, add to the config. file

$period=10;

By default kiko> enforces that a given number of processes under its control equal to **$ncpu** are running. Other possibilities are to examine the process table in order to judge whether more processes should be launched (by setting **$watch='ps';** in the config. file) or to base the decision on the computer's load, as provided by the *uptime* command (**$watch='load';**). Using the number of kiko> jobs as dispatching criterion allows very high speeds (<<1 sec), as there is no need to make constant inquires to the system. Using **ps** to evaluate the system load will be quite stable. Both **ps** and **load** will readily accommodate other jobs, perhaps interactive ones, which may not be running under kiko>, at a high priority, but because *uptime* provides averages over 1 or more minutes, a minimum $period of 30 seconds is required for the **load** option.

4.6. help

A quick list of the available commands in kiko> can be printed by typing

$kiko –-help

and version information is available using

$kiko –-version

All of the commands that are preceded by '–-' above can also be specified without '–-'. E.g.

$kiko help

5. FAQs

Why should I use kiko> instead of any other available job queueing software?

Kiko> is very simple and lightweight. If all you need is to handle your own jobs on your own workstation/desktop/laptop, then kiko> will probably satisfy your needs -- it will be the easiest and fastest to set up and learn.

Kiko> executes the jobs in the same order they are submitted to the queue: there is no optimization. If your tasks are all balanced, i.e. they can keep a single core busy and use less than 1/n of the system's RAM memory (where n is the number of cores available), and especially if

most of them are of the same or similar length (time duration), you will likely be happy with kiko>.

What are kiko>'s dependencies?

Not many. If you have a unix-like OS (e.g. any flavor of linux, MacOS X, or Solaris) , you're ready to use kiko>. Kiko> is written in Perl, and Perl 5 is usually part of any unix-like system these days.

Is there a graphycal user interface for kiko>?

I'm afraid not.

Can I use kiko> on distributed clusters?

Sorry, not for now. Kiko> was written with shared-memory computers, like common desktops or laptops, in mind.

Is kiko> free?

Kiko is publicly available under the GNU General Public License, included in the distribution. That means you can download and use kiko> incurring no charges, but there are some constraints regarding how to change it, copy, or redistribute it. See the LICENSE file, or point your browser to
http://www.gnu.org/licenses/gpl-3.0.html

What's the picture in the cover?

A flock of pelicans flying near San Diego, California, in Fall 2007.

5. most recent changes in the manual/behavior

Since v1.2 the default directory for submitting jobs is that of the executable; see Section 4.2.

6. modification history

v1.0 written in July 2009

v1.1 added a check to make sure the deamon pid is well and kicking instead of trusting the semaphore file blinding, August 2009

v1.2 modified the default dir from ‘pwd’ to where the executable’s path; now ‘kiko hyd*/job.bash’ will successfully submit the job.bash scripts from
all the local hyd* subdirs, and still works with commands, as they are handled as scripts in the local (pwd) dir, October 2009