Gmail **wruslandr wyusoff <wruslandr@gmail.com>**

WRY-UMP-PhD SUCCESS Asynchronous (parallel) versus Synchronous (sequential) runs using Julia

2 messages

wruslandr wyusoff <wruslandr@gmail.com> Sun, Aug 25, 2019 at 2:57 AM
To: wruslan wyusoff <wruslandr@gmail.com>, Fadhlur Rahman Bin Mohd Romlay <fadhlur@ump.edu.my>, "fadhlur.ump" <fadhlur.ump@gmail.com>, yashwant prasad singh <ypsingh45@gmail.com>

Why asynchronous/parallel? Because we are looking at opportunities in CNC where we can parallelize some parts of the execution. Ha ha ha. See the attached flowcharts as screenshots.

This julia code (in the zipped file) compares the synchronous and asynchronous execution times (run_durations) of a common shared function named "do_same_work(parameters)". There are four(4) tasks to be executed. Each task will perform the same function, do_same_work(parameters). However, each task will be provided with different input parameters. In this code, we decided on four(4) tasks because we have physically 4-CPU cores on our notebook.

In Julia, the declaration of tasks using macros @async and @sync on a function launches (executes) the task immediately. We can think of these macros as task launchers instead of just function definitions..

Example synchronous tasks:

```
task01 = @sync do_same_work("task01", 100, 10, 0.3, 5);
task02 = @sync do_same_work("task02", 10, 10, 0.1, 10);
task03 = @sync do_same_work("task03", 50, 5, 0.2, 4);
task04 = @sync do_same_work("task04", 0, 10, 0.5, 7);
```

Example asynchronous tasks:

```
task11 = @async do_same_work("task11", 100, 10, 0.3, 5);
task12 = @async do_same_work("task12", 10, 10, 0.1, 10);
task13 = @async do_same_work("task13", 50, 5, 0.2, 4);
task14 = @async do_same_work("task14", 0, 10, 0.5, 7);
```

RESULTS EXTRACTED:

Total synchronous (sequential) run_duration = 26.24190902709961 sec.
Total asynchronous (concurrent) run_duration = 10.42530083656311 sec.

(1) In the synchronous (sequential) execution, all four(4) tasks will be executed sequentially (one after another) and the accumulated (additive) total time will be the synchronous run_duration. In this mode, the running sequence is (task01 -> task02 -> task03 -> task04) and finish. The individual task durations are additive. From the results above, the total run_duration to finish is about 26 seconds.

(2) Asynchronous running tasks means independent running tasks, that is, once started each task runs to completion without interruption from any other event. In this asynchronous (concurrent) execution, all four(4) tasks will be executed simultaneously (task11, task12, task13 and task14). In this mode, all four(4) tasks execute in overlapping time, with one task on each CPU core. Since the tasks are running in parallel, the net completion time taken is the time to finish the longest task. The times are overlapping and are not additive, meaning the earlier tasks have already finished. From the results above, the longest task took about 10 seconds.

(3) For demonstration purposes, we uncommented a print statement to show details of task executions in overlapping time. The normal run is not to display this line. From our results provided at the end of this code, we get overlapping runs below:

Time range (41.41 - 45.65) sec. All tasks running. Tasks task11, task12, task13 and task14 are running in parallel.
Time range (45.65 - 46.53) sec. Task13 finished. Tasks task11, task12 and task14 continue running in parallel.
Time range (46.53 - 48.74) sec. Task11 finished, Tasks task12 and task14 continue running in parallel.
Time range (48.74 - 51.56) sec. Task14 finished. Task12 continue running alone to completion.

SOME NOTES.

(N1) Putting macro @async and @sync (on function) immediately and automatically start both task executions.

(N2) Because we display high resolution time in the outputs, we display the start and finish times for all of the four(4) tasks, covering both synchronous and asynchronous execution.

(N3) We also can sort the results display according to this time.

(N4) To prove that the calculations are correct, all tasks running in synchronous and asynchronous modes gave the same answers.

(N5) In the asynchronous case, we can print the time details of the individual tasks (taskname) as they happen, so we can see the overlapping task executions.

WHY WE USE JULIA (Runs fast like C) For testing use julia, then convert to C-Code.
==========
Launched in 2012, Julia is an open-source programming language that combines the interactivity and syntax of scripting languages, such as Python, Matlab and R, with the speed of compiled languages such as Fortran and C.

Julia tagline:
Julia looks like PYTHON, feels like LISP and runs like C or Fortran.

Julia speed:
Julia is essentially a compiled language in a scripting-language clothing. Julia runs like C, but reads like Python. Jane Herriman, who is studying materials science at the California Institute of Technology in Pasadena, says that she has seen ten-fold-faster runs since rewriting her Python codes in Julia.

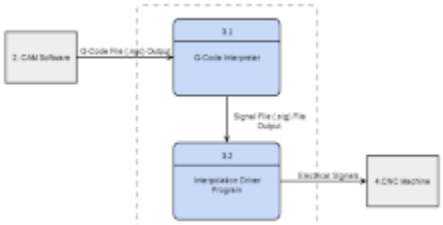
Julia parallel:
Julia includes built-in features to accelerate computationally intensive problems, such as distributed computing (parallel), that otherwise require multiple languages. (Distributed computing allows programmers to split difficult problems across multiple processors and computers.)

Julia features:
Julia provides interesting features such as multiple dispatch (allowing multiple functions to have the same name) and metaprogramming (programs that can modify themselves).

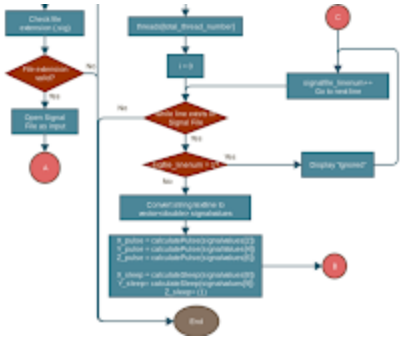
Read about Julia in the PDF attachment in the zipped file.
--
WASSALAM
wruslandr

---

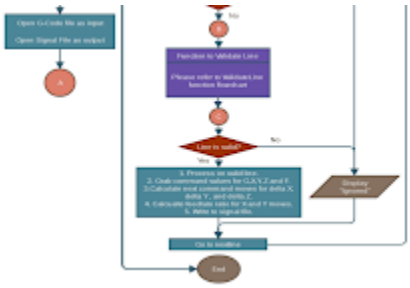
4 attachments



Screenshot at 2019-05-21 10-58-55.png
46K



Screenshot at 2019-05-21 11-05-33.png
114K



Screenshot at 2019-05-21 11-00-24.png
107K

SUCCESS-Asynchronous-Synchronous-codes-in-julia.zip
329K

---

yashwant prasad singh <y.p.singh@mmu.edu.my> Sun, Aug 25, 2019 at 8:18 PM
To: wruslandr wyusoff <wruslandr@gmail.com>

julia is programming language may have support for parallel asynchronous or synchronous computing framework, but is developed for scientific computing, very similar to python. My understanding still suggests C/C++ is still the best choice for real-time machine control for industrial applications.

"**Julia** is a high-level programming language designed for high-performance numerical analysis and computational science"

With thanks,
Y.P.Singh
[Quoted text hidden]